NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DESIGN AND IMPLEMENTATION OF A
DEBUGGER FOR AN ABSTRACT MACHINE

by

Stanley Victrum

June 1987

Thesis Advisor: Daniel L. Davis

REPORT DOCUMENTATION PAGE

a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited
b DECLASSIFICATION / DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION Naval Postgraduate Schoo	6b OFFICE SYMBOL (If applicable) Code 52
7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION	8b OFFICE SYMBOL (If applicable)
9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
DESIGN AND IMPLEMENTATION OF A DEBUGGER FOR AN ABSTRACT MACHINE (u)

12 PERSONAL AUTHOR(S)
ictrum, Stanley

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
aster's Thesis	FROM _____ TO _____	1987 June	138

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Interactive debugger; Software portability; Resource abstraction; Formal specifications; Abstract machine; Representation independence;

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Conventional computer architectures do not allow us to unambiguously xpress our intent in a computer program. The combination of artificial ata types and resource models force ambiguity and data structure over- oading. For example, the semantics of a stack combine those of an array tructure and a last-in-first-out queue, while the entire stack structure s implemented in computer memory as a group of fixed length cells. This nd other machine-data type dependencies can markedly hamper software ortability. To overcome these obstacles, a means of formally specifying computing machine's physical resources in an implementation independent ay has been proposed. Creating an abstraction of the computer's physi- al resources in this manner lets the implementor of the specifications learly determine the intent of programs written for it. This abstrac- ion has come to be known as the Abstract Machine or AM.

20 STRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL aniel Davis	22b TELEPHONE (Include Area Code) (408) 646-3091
	22c OFFICE SYMBOL Code 52Dv

D ORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

#18 SUBJECT TERMS (continued)

Functional interfaces.

#19 ABSTRACT (continued)

One implementation of these resource specifications has already been accomplished. Several programming tools, such as a programming language compiler and a visual display device, have also been created (in software) for use with this AM's implementation. At present, however, there are no means for interactively displaying and altering the storage resources of the Abstract Machine for debugging purposes. For the current AM implementation, the bulk of the automated debugging tools consist of assembler code tracing and listing options that can be chosen at run time. The goal of this thesis is to build an interactive debugger for the Abstract Machine near the assembler code level. This should expedite the process of producing relatively error-free, executable programs while using a smaller amount of time and effort. The debugger will serve as another building block in the creation of a complete programming environment for the Abstract Machine. This in turn will assist in the general study of minimizing the software portability problems that arise because of machine-software dependencies.

Design and Implementation of a Debugger for an Abstract Machine

by

Stanley Victrum
First Lieutenant, United States Marine Corps
B.A., The Citadel, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1987

ABSTRACT

Conventional computer architectures do not allow us to unambiguously express our intent in a computer program. The combination of artificial data types and resource models force ambiguity and data structure overloading. For example, the semantics of a stack combine those of an array structure and a last-in-first-out queue, while the entire stack structure is implemented in computer memory as a group of fixed length cells. This and other machine-data type dependencies can markedly hamper software portability. To overcome these obstacles, a means of formally specifying a computing machine's physical resources in an implementation independent way has been proposed. Creating an abstraction of the computer's physical resources in this manner lets the implementor of the specifications clearly determine the intent of programs written for it. This abstraction has come to be known as the Abstract Machine or AM.

One implementation of these resource specifications has already been accomplished. Several programming tools, such as a programming language compiler and a visual display device, have also been created (in software) for use with this AM's implementation. At present, however, there are no means for interactively displaying and altering the storage resources of the Abstract Machine for debugging purposes. For the current AM implementation, the bulk of the automated debugging tools consist of assembler code tracing and listing options that can be chosen at run time. The goal of this thesis is to build an interactive debugger for the Abstract Machine near the assembler code level. This should expedite the process of producing relatively error-free, executable programs while using a smaller

amount of time and effort. The debugger will serve as another building block in the creation of a complete programming environment for the Abstract Machine. This in turn will assist in the general study of minimizing the software portability problems that arise because of machine-software dependencies.

TABLE OF CONTENTS

I. INTRODUCTION 9
II. BACKGROUND 11
A. THE PROBLEM 11
B. A FEASIBLE SOLUTION: AM 13
C. RELATED RESEARCH 14
III. DESIGN 15
A. BASIC COMPUTER RESOURCE ORGANIZATION 15
1. Computer Resources 15
2. Computer Data Structures 17
3. Instructions to the Computer 19
B. DESIGN OF THE INTERACTIVE DEBUGGER 20
1. Philosophy 20
2. Debugger Interface to the Physical Resources 23
3. Debugger Interface to the User 24
IV. IMPLEMENTATION 26
A. ASSUMPTIONS 26
B. DEBUGGER COMMAND SYNTAX SYMBOLS 26
C. INTERACTIVE DEBUGGER OPERATIONS 26
1. Display Operations 26
a. Display Memory 27
b. Display Register 27
c. Display Stacktop 28
d. Display Breakpoints 28
e. Display Program Counter 29
2. Set Operations 29
a. Set Memory 29
b. Set Register 30
c. Set Stacktop 30
d. Set Breakpoint 31
e. Set Program Counter 32
3. Remove Breakpoint Operation 32
4. Trace Execution Operations 33

a. Trace On 33
b. Trace Off 33
c. Trace n 34
5. Go Operations 34
a. Go Uncontrolled 34
b. Go n 35
6. Help Operation 35
D. DEBUGGER CONTROL OF MACHINE EXECUTION 35
1. Debugflag 36
2. Debgtask 36
3. Left2do 36
4. Debgcntl 36
5. Breakflg 38
6. Errorflg 38
7. Dbgtrace 38
E. ERROR HANDLING IN PROGRAM EXECUTION 38
F. MODIFICATIONS TO AM IMPLEMENTATION 39
V. CONCLUSIONS AND FUTURE WORK 40
APPENDIX A: SAMPLE SESSIONS 41
APPENDIX B: DEBUGGER COMMAND SYNTAX SYMBOLS 70
APPENDIX A: DEBUGGER PROGRAM FILES 71
LIST OF REFERENCES 135
INITIAL DISTRIBUTION LIST 136

LIST OF FIGURES

Figure 2.1 : The Semantic Gap 11

Figure 2.2 : Data Type Dependency on Hardware Values 12

Figure 2.3 : Narrowing the Semantic Gap 14

Figure 3.1 : Computer Program in AM Implementation 17

Figure 3.2 : Data Value Structures 18

Figure 3.3 : Common Value Abstraction 19

Figure 3.4 : Primary Resource Structures 20

Figure 3.5 : Structure of Instruction Value 21

Figure 4.1 : Debugger Control Variables 37

I. INTRODUCTION

In the days when the assembly language programmer was king and higher-level programming languages were still unimplemented concepts on the "drawing board," crude were the tools the programmer could wield in his programming environment to quickly conquer coding problems. During this period, CPU time was money, thus efficient programs were a must (as well as a source of pride and means of security for the programmer). The programmer was expected to massage the computing machine's stacks, registers and memory to eke out as efficient a program as possible.

To write efficient programs, the programmer typically included in his program code assumptions about his computing machine's physical resources. These assumptions, such as the number of registers available, the machine's representation of data types, and the physical implementation of the stacks, were "hardwired" into the programmer's code. Of course, errors in the program required intimate knowledge of all these assumptions. Events such as upgrading the machine or replacing it with one of a slightly different architecture typically caused program nightmares with previously "bug-free" programs going haywire because the resource assumptions had changed. This situation, unfortunately caused even more assumptions to be incorporated into the code, typically in the form of program "patches." This cycle, if allowed to continue, can so disfigure the original intent of the program, it soon becomes difficult, at best, to interpret it.

Conventional computer architectures do not allow us to unambiguously express our intent in a computer program. The combination of artificial data

types and artificial resource models force ambiguity and data structure overloading. For example, the semantics of a stack combine those of an array structure and a last-in-first-out queue, while the entire stack structure is implemented in computer memory as a group of fixed length cells. This and other machine-data type dependencies can markedly hamper software portability. To overcome these obstacles, a means of formally specifying a computing machine's physical resources in an implementation independent way has been proposed (Davis[1984]). Creating an abstraction of the computer's physical resources in this manner lets the implementor of the specifications clearly determine the intent of programs written for it. This abstraction has come to be known as the Abstract Machine or AM.

One implementation of these resource specifications has already been accomplished (Yurchak[1984]). Several programming tools, such as a programming language compiler (Ozisik[1986]) and a visual display device (Hunter[1985]), have also been created (in software) for use with this AM's implementation. At present, however, there are no means for interactively displaying and altering the storage resources of the Abstract Machine for debugging purposes. The current debugging tools consist of assembler code tracing and listing options that can be chosen only at run time. This thesis' goal is to build an interactive debugger for the Abstract Machine near the assembler code level. This should expedite the process of producing relatively error-free, executable programs while using a smaller amount of time and effort. The debugger will serve as another building block in the creation of a complete programming environment for the Abstract Machine. This in turn will assist in the general study of minimizing the software portability problems that arise because of machine-software dependencies.

II. BACKGROUND

A. THE PROBLEM

In his Masters thesis, Yurchak[1984] presented a formal specification for an Abstract (computing) Machine which he called AM. This AM was to be used to study and offer a way of minimizing the problem of porting software from one computing machine to another.

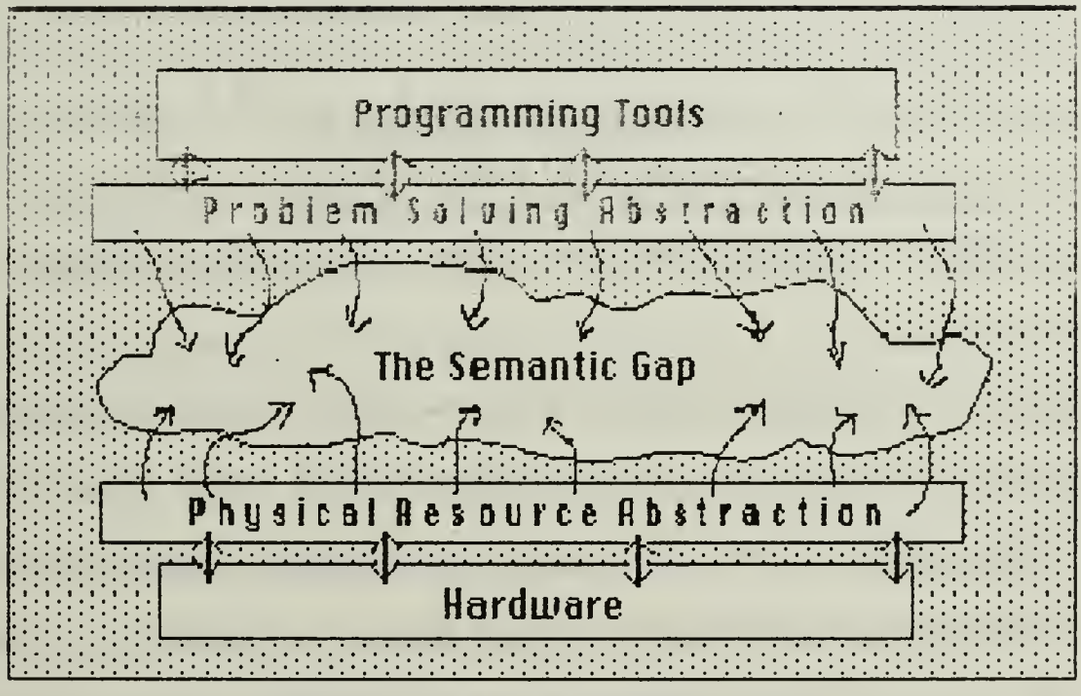

Figure 2.1: The Semantic Gap

Yurchak[1984] noted that porting large programs between computing machines is an expensive ordeal in terms of programmer time and effort. This predicament is brought on because of the wide semantic gap (as shown in Figure 2.1 above) between the programmer's problem solving abstraction (i.e. programming languages, development tools, etc.) and the computing machine's physical resources abstraction (i.e. addresses, registers, stacks,

etc.). In his words, this gap between the two abstractions was, simplistically speaking, a "boundary" between the software used by the programmer to form problem solutions and the hardware by which those solutions are implemented.

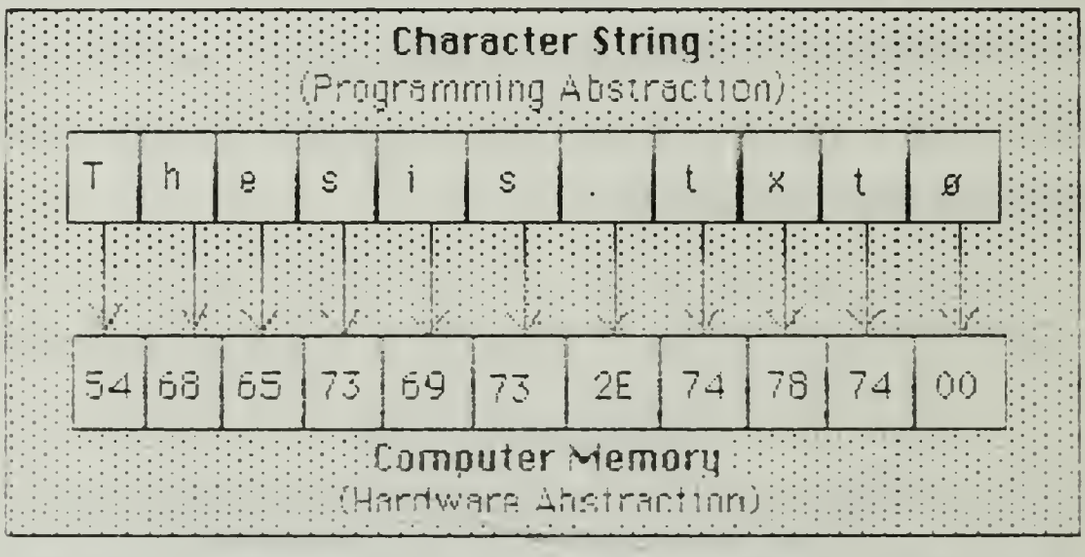

Figure 2.2: Data Type Dependency on Hardware Values

Generally, a computing machine has but a few primitive structures on which the rest of the problem solving abstraction is based. Typically, there exists a strong bond between the way data types are implemented and the manner in which they are represented in hardware. For example, a character string is typically represented as an array of memory cells, with each cell containing the corresponding integer representation of the corresponding character in the character string. Figure 2.2 above depicts this relationship. If the textual representation of the values in computer memory were to be removed, the meaning of the values could not be determined. They could be integers, memory addresses or numeric symbols for other artificial data types. The context of the values must be determined before they can be given meaning. This situation typically leads to overloading of the primitive data

types of the machine, allowing the programmer to treat one data type as another, thus compromising the data structure typing. This compromise typically causes extensive program changes in the data structure definitions when porting the program from one architecture to another.

B. A FEASIBLE SOLUTION: AM

In proposing a solution to the software portability problem, Davis[1984] formulated a methodology for formally specifying a computer's resources at different levels of abstraction. His approach was to develop formal specifications for the functional interfaces between the resources of the computing machine. In this way, the user need only be concerned with the functional interface to a particular abstraction level and not the actual interface implementation, which could be done in hardware or software. To ascertain the feasibility of this new methodology, Yurchak[1984] designed and implemented a test version of the specifications for a computer processor, which has become to be known as the Abstract Machine. Davis[1984] and Yurchak[1984] decided to test the methodology at the processor level of abstraction because, being the most difficult to formally describe, it would give the most insight on the validity of the approach.

This abstract architecture treats each one of the machine's physical resources as a black box and allows the programmer to use the resources in only the specified way. In other words, the specification details exactly what resources mean and how to use them, but does not specify how a resource is to be implemented. With software tools being written for implementation on one abstract machine, software portability is markedly improved. The AM serves as a formal interface between the programming

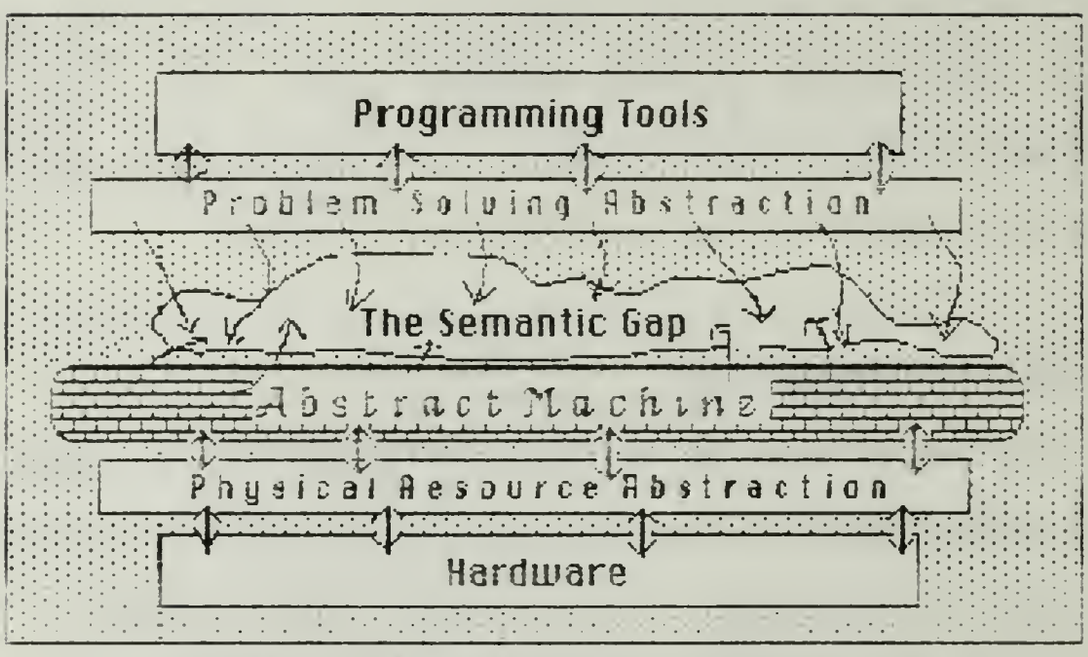

Figure 2.3: Narrowing the Semantic Gap

tools in the environment and the physical resources of the computing machine. The AM programming tools will work on any architecture as long as the AM specifications are properly implemented on that architecture. Figure 2.3 above depicts the AM interface in the programming environment.

C. RELATED RESEARCH

Several programming tools have already been developed for the AM. Hunter[1985] formally designed and specified a visual display device for the AM. Ozisik[1986] designed and implemented a subset C compiler which produces AM assembly language code. Zang[1985] formally specified and designed an abstract database using similar principles for specifying the AM. Again, the goal of this thesis is to create another development tool for the AM programming environment, namely, an interactive debugger near the assembly code level.

III. DESIGN

Before presenting the design and implementation of the AM interactive debugger, it is important to make clear the salient points behind the specification of the AM. The AM is an abstraction of the physical resources of a machine and, as Davis[1984] points out, the methodology used to formally specify that abstraction is *representation independent*. The best example of this representation independence notion is abstract data types. The type 'integer' not only implies a set of values, but a set of operations upon those values. This "notion" can be implemented in many different contexts, but its intuitive properties transcend implementations or physical representations. This is the essence of and the true power behind the Abstract Machine concept.

The actual AM implementation around which the interactive debugger was built is another physical representation of this representation independence notion. Therefore, although one implementation is presented in this thesis, the formal specifications for the resource abstraction is adhered to (via the programmer interface to the debugger).

A. BASIC COMPUTER RESOURCE ORGANIZATION

1. Computer Resources

The AM incorporates the basic principles of a von Neumann computing machine. It has a memory for program storage and instruction execution, sets of registers and stacks for temporary storage of data values, and, of course, a set of data values. Yurchak's[1984] AM implementation, being an

abstraction of a computing machine's physical resources, can be easily "reconfigured" by changing the basic definitions of the resources available. The implementation storage resources used in developing the debugger were as follows:

- Two memory segments, each with 1024 storage cells;
- One register segment with thirty-two storage cells;
- One stack segment with 512 storage cells;
- 1022 heap segments, each with 1024 storage cells.

The memory, registers, stacks and heap storage cells, as in any computing machine, all hold defined data values. The definition of AM data values, however, differs slightly than the manner in which they are defined in other computing systems. The AM's data values are typed while a regular von Neumann machine's is not. In other words, one can determine what the value is inside a storage location since the value's type is stored along with it. Figure 3.1 on page 17 shows an example of a computer program in this AM implementation with its machine code and assembly language statements. At memory location '00000000,' '0190' is the AM implementation's machine code for INSTRUCTION-TYPE. It is followed in the memory address by the instruction opcode and the operands for the opcode. Notice that each operand value is also linked to a type. '0160' is machine code for MEMORY-ADDRESS-TYPE and '0170' for REGISTER-ADDRESS-TYPE.

The AM physical resources abstraction, being implemented in software, is built upon data structures. The data structures used by Yurchak[1984] were studied in depth so that the computer's functional capabilities could be extended and, thus, are presented in the next section.

Memory Location	Machine Code			Assembly Language Statements	
	0190	18F0			
00000000	0190	3831		move	{addr, data}, r (0:0)
	0160	00000003			
	0170	00000000			
00000001	0190	382E		move	{int, 100}, r(0:0)@
	0130	00000064			
	0170	00000000			
00000002	0190	1880		stop	
00000003	0190	18F4	data	ds	10
	0160	0000000D			
	0190	18F1			

Figure 3.1: Computer Program in AM Implementation

2. Computer Data Structures

In order to design a debugger that will interact unobtrusively with the AM's basic operations, one must understand the data structures upon which it was implemented. These structures represent the AM's data values, memory, registers, and stacks. All the data structures used in this implementation were written in the 'C' programming language.

Figure 3.2 on page 18 shows the type declaration structures in Yurchak's[1984] implementation for BOOLEAN, INTEGER and NATURAL data values. They are representative of all the AM's basic data value structures. Yurchak[1984] notes that the AM storage resources are designed to hold any properly defined value. In typical computing, this poses little or no problem since all the values are based upon the overworked and overloaded bit vector. To be able to store and operate upon a myriad of value types in the

```
typedef char            bool;
typedef unsigned int    nat;

typedef struct {
        short  type;
        bool   val;   } BOOL;

typedef struct {
        short  type;
        long   val;   } INT;

typedef struct {
        short  type;
        nat    val;  } NAT;
```

Figure 3.2: Data Value Structures

AM's "physical resources" required the introduction of another "common" level of abstraction, a union of all the basic value types. This abstraction was implemented using the structure shown in Figure 3.3 on page 19.

With the structures of the data values now presented, it is now time to examine the structures that represent this implementation's primary physical resource abstraction for the memory, registers and stacks. The type declarations for the structures are shown in Figure 3.4 on page 20. Each resource is primarily a structure containing an array which can store any value defined in this AM implementation. The computer's memory, registers and stacks are actually arrays of these resource structures. Each of these arrays equates to a storage "segment."

```
typedef   union     value {
          short     type;
          opcode    opcdval;
          BOOL      boolval;
          INT       intval;
          NAT       natval;
          CHAR      charval;
          CSTR      cstrval;
          MAD       madval;
          RAD       radval;
          SAD       sadval;
          FIL       fileval;
          INSTR     instrval;
          MOP       mopval;
          DOP       dopval;
          ROP       ropval;
          BOP       bopval;   } VAL;
```

Figure 3.3: Common VALUE Abstraction

3. Instructions to the Computer

Perhaps the most important of the computer values is the INSTRUCTION. INSTRUCTION values, as in any other computing machine, drive the computer program's execution. Its type definition and logical structure in Yurchak's[1984] implementation are shown in Figure 3.5 on page 21. As shown in the figure, the instruction is implemented as a structure containing its value type and an array of operand values. The first element of this array is the instruction opcode. Subsequent elements in the array are the operands

```
typedef struct {
        int     size;
        VAL     **val;  } memseg;

typedef struct {
        int     num;
        VAL     **val;  } regseg;

typedef struct {
        int     size;
        long    sp;
        VAL     **val;  } stkseg;
```

Figure 3.4: Primary Resource Structures

(AM data values) for the instructions. Yurchak[1984] implemented the instruction value so that the first digit of the opcode indicated the number of operands the instruction required. By his design, the opcode was also considered as one of the operands. It is important to note that the instruction type definition (Figure 3.5) and the common-value type definition (Figure 3.3) are recursively defined in terms of each other. This facilitates converting from a basic instruction value, stored in memory, to an executable instruction at the programming level of abstraction.

B. DESIGN OF THE INTERACTIVE DEBUGGER

1. Philosophy

As stated in Chapter I, the computer's current debugging facility consists of a trace option that can be specified at computer "startup." This

```
typedef  struct {
         short    type;
         union    value  *val;  } INSTR;
```

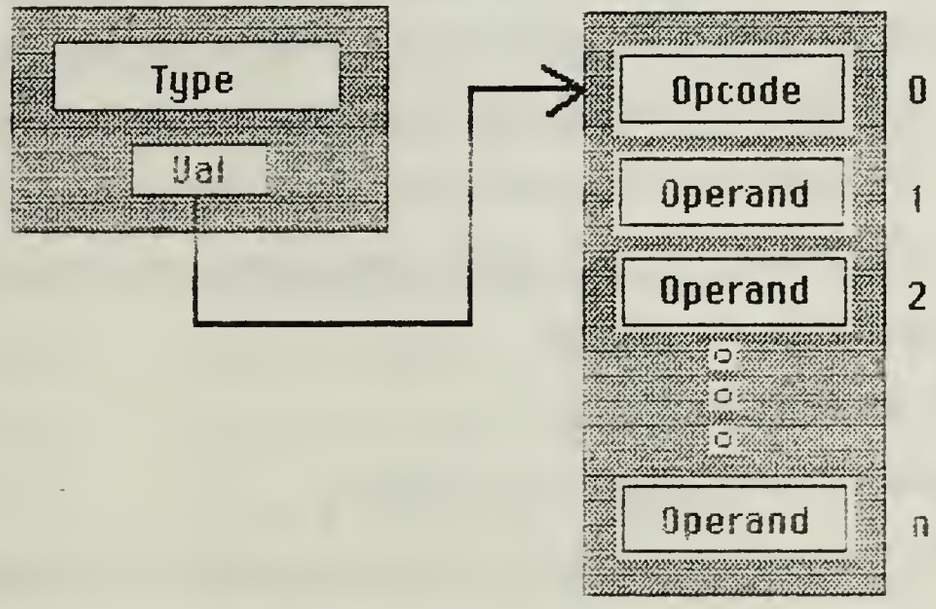

Figure 3.5: Structure of Instruction Value

trace facility was, however, designed more for providing debugging diagnostics about the computer's internal functioning than for debugging the user's program. The trace could only be turned on at the beginning of program execution and could only be turned off by program termination. This facility alone clearly does not provide the AM computer programmer with an adequate program debugging environment.

Wray[1984] notes that a microcomputer debugging environment should have the following basic functions:

- Single-step program execution;
- Breakpoints in program execution;
- Register Display/Modification;

- Memory Display/Modification.

This basic suite of debugging facilities gives the programmer the capability to "disassemble" and examine his program and the machine state in discrete, well defined steps. This gives the programmer a means of quickly identifying and correcting program logic errors with a minimum of time and effort. The interactive debugger for the AM was designed to provide all of the above facilities. To give the programmer an added degree of freedom in the debugging environment, the following capabilities were also incorporated into the design of the AM debugger:

- Program Execution Trace;
- Program Counter Display/Modification.

The program execution trace, unlike the old trace facility, shows just the instruction about to be executed. Allowing the programmer to see and change the program counter allows the testing of different program execution paths without having to terminate program execution, changing its textual representation, and recompiling the program for execution. This is in accord with the philosophy of providing a debugging environment that facilitates quick detection and correction of coding (logic) errors.

Yurchak's[1984] AM implementation is designed to be easily reconfigurable by changing the basic array definitions of the resources available. The essence of this design objective was carried over into the design of the debugger. A reconfiguration of the computer's resources in the AM is acknowledged by the debugger while performing its operations. The basic definitions for the debugger may also be easily reconfigured for, say, allowing more program breakpoints or increasing the maximum integer the debugger assumes.

2. Debugger Interface to the Physical Resources

In order to implement the debugger facilities, an interface to the computer's physical resources had to be designed. This interface would be primarily concerned with the retrieval and storage of data values in the machine's physical resources. A study of the computer's data retrieval and storage modules revealed that the following functions were currently available for these tasks:

- FETCHM() - retrieves a value from memory;
- STOREM() - stores a value into memory;
- FETCHR() - retrieves a value from a register;
- STORER() - stores a value into a register;
- TOPSTK() - retrieves a value at top of a stack.

These functions provide a well-defined interface to the data structures upon which the resource abstraction is built. Interfacing to these functions to perform the debugger operations resulted in a reduction in the amount of code needed to implement the debugger.

As is typical in any computing machine, however, some data retrieval and storage operations generate errors which in turn force abnormal termination of program execution. Three such operations are as follows:

- Retrieving a value form an uninitialized storage location;
- Retrieving a value form a non-existing storage location;
- Storing a value at a non-existing storage location.

Clearly if the debugger is to use the implementation's existing AM data retrieval and storage functions to perform its operations, a means of properly handling these types of errors had to be developed. To display uninitialized storage locations while in the debugging mode, the retrieval

functions were modified to return a NULL value for displaying to the user. If the computer is not in the debug mode, a regular execution error is generated. The means for preventing the latter two types of errors while in the debug mode were designed into the user interface to the debugger.

3. Debugger Interface to the User

The interface was designed to permit, of course, the capability to perform the operations listed in section III.B.1. It was also tailored to keep the user from specifying debugger commands that would cause preventable errors in the debugger interface to the physical resources abstraction and in program execution. Since the debugger has access to available computer resources, the user can be kept from trying to access a non-existing storage segment or offset address, from setting the program counter to a memory location that does not contain an instruction value, or setting a breakpoint at a memory location that does not contain an instruction value. This type of interface traps potential errors at the earliest possible stage. It does not, however, prevent the user from setting a storage location with a value that may cause an error during program execution. This particular kind of error correction was considered beyond the scope of this thesis and, therefore, was not entertained.

The debugger was designed to prompt the user for each piece of command input (prompted commands) instead of the user entering the entire debugger command on one or two lines (line commands). Prompted input allows for "layered" error checking of the debugger command, allowing the user to reenter input at that layer instead of having to reenter the entire debugger command. This also permits new users, familiar only with this AM implementation's value representation and instruction opcodes, to quickly

learn and use the debugger without learning debugger command-line formats. The debugger interface to the user is shown in Appendix A, Sample Sessions.

IV. IMPLEMENTATION

A. ASSUMPTIONS

In building any software tool, some of the many variables in a programming environment must be made fixed due to implementation considerations and a need for establishing a point of reference. These "constant" variables take the form of assumptions about the programming environment. The following ones have been made about the AM programming environment:

- The user knows how to assemble files using the environment's assembler for execution on the computer;
- The user is familiar with the instruction opcodes used in this implementation of the AM;
- There are no more than six (6) operands per instruction;
- A character string is less than 81 characters in length;
- Integer values range from - 2147483647 to 2147483647;

B. DEBUGGER COMMAND SYNTAX SYMBOLS

The debugger command syntax symbols are in Appendix B, Debugger Command Syntax Symbols. The user is prompted for each piece of the command, therefore, the actual command format is relatively unimportant.

C. INTERACTIVE DEBUGGER OPERATIONS

1. Display Operations

The user can use the debugger to display all the computer's memory,

registers, the top of any stack segment, all the program breakpoints and the current value of the program counter. Sample demonstrations of the display operations can be found in Appendix A, Sample Sessions. The following subsections present the functional details for each type of display operation.

a. Display Memory

All of the computer's memory cells can be displayed with the 'display memory' operation. The command has the following syntax:

'd' 'm' {'*' | segment:offset} span

The operation retrieves a value from a specified memory cell and displays it to the user. It uses the computer function 'fetchm()' to perform the retrieve and 'showmem()' to display the value at the memory location. Normally, during regular program execution, a retrieval from an uninitialized memory location causes an execution-terminating error in the computer. For debugging purposes, however, a means was developed to interject a null value into the retrieve if, in fact, no value is contained at the specified memory cell. A control variable was used to tell the computer whether to return a null value (implying the machine was performing a debugger task) or generate an error (machine under program execution). Control variables are covered in section IV.D.

b. Display Register

All of the computer's registers can be viewed with the 'display register' operation. It has the following command syntax:

'd' 'r' segment:offset span

Like the 'display memory' operation, this operation causes similar retrieval

and display operations to be performed, but using the AM's register segments instead of the memory segments. This operation uses 'fetchr()' to retrieve the value from the register cell and 'showmem()' to display the value to the user. As in displaying memory, similar provisions are made for the interjection of a null value if the register cell contains no value.

c. Display Stacktop

The top of any stack segment can be displayed using the 'display stack' operation. Its command syntax is as follows:

'd' 's' segment

The mechanics of the 'display stack' operation are similar to that of 'display memory' and 'display register'. It, however, uses 'topstk()' to retrieve the value stored at the top of a stack segment. One might wonder why the user is given a free hand in viewing any memory or register cell, but is restricted to seeing only the top cell in a stack segment. This, in part, is in keeping with the notion that the resources are a "black box." For a particular state of the computer, the values stored in the memory and registers 'exist' and are all accessible by the user, typically via a computer program. Values in stack cells below the top cell conceptually do not exist for a particular state of the machine. The interactive debugger adheres to this principle.

d. Display Breakpoints

This operation displays all the entries in the breakpoint table. The command syntax is as follows:

'd' 'b'

The table has three items per entry: the break number, the memory location

where the breakpoint is set, and the opcode of the regular instruction. The table is updated by the 'set breakpoint' and 'remove breakpoint' commands.

e. Display Program Counter

This debugger operation displays what segment the program counter is currently in and at what offset in the segment it is currently pointing. Its command syntax is as follows:

'd' '*'

2. Set Operations

The set operations are perhaps the most important of all the other debugger operations since they permit the user to actually change the state of the computer. The operations give the use the capability to alter the state of the computer's memory, registers, stacktops and the program counter. The set operations also include the capability to set breakpoints at specified memory locations. Demonstrations of the set commands can be found in Appendix A, Sample Sessions. The functional description of each of the set operation now follows.

a. Set Memory

The debugger operation 'set memory' gives the user the capability to store any of the AM's defined values at any memory location, uninitialized or not. This allows the user to "patch" faulty instructions so that program testing can proceed upon a user-desired path. Its command syntax is as follows:

's' 'm' segment:offset value

This operation uses the computer function 'storem()' to place the desired AM

value into a specified memory location. This operation can indirectly affect the behavior of another debugger operation, 'remove breakpoint' (which is covered in detail later). Breakpoints are implemented by substituting breakpoint opcodes for the actual instruction opcode in the instruction value. Setting a memory location which contains a breakpoint opcodewould, in effect, remove the breakpoint opcode,but leave its breakpoint entry in the break table. For this reason, the 'set memory' operation first retrieves the value using 'fetchm()' and checks to see if a breakpoint exists at the location. If there is, the user is given the option of aborting the operation or confirming it. If the operation is confirmed, the breakpoint is removed from the table and the new value stored into memory. This technique helps to ensure closure in the debugger operations and to maintain a significant degree of operation independence between the debugger commands.

b. Set Register

This operation allows the user to set any register location to any one of the regularly defined values. Its command syntax is as follows:

's' 'r' segment:offset value

It uses the function 'storer()' to store 'value' into the indicated register.

c. Set Stacktop

This operation lets the user store any regularly defined value at the top of any stack segment. Its command syntax is as follows:

's' 's' segment:offset value

This AM implementation contained no function for actually changing the value at the stacktop without modifying the stack pointer. In other words,

the value at the stacktop can normally be changed only by pushing values onto or popping values off of the stack, thus changing the stack pointer. To give the user the freedom to actually alter the value at the stacktop without modifying (in all but one case) the stack pointer, a new function was added to the computer called 'storestk().' The function uses the stack segment stack pointer to actually store the value at the stacktop. It does change the stack pointer when the stack is empty since the stack pointer must be initialized before the value can be stored.

d. Set Breakpoint

This operation gives the user the capability to temporarily halt program execution to examine and possibly alter the state of the AM. Its command syntax is as follows:

's' 'b' segment:offset

Breakpoints can be set in any memory location that contains an instruction value. The operation will self-abort if a non-instruction value is stored at the memory address (and the user is so informed). The operation also self-aborts if a breakpoint is currently set at the memory location.

The operation first uses 'fetchm()' to retrieve the value stored at the specified location, Then the aforementioned tests are performed. If the value in memory is an instruction and no breakpoint is set at the location, a new breakpoint opcode is computed, the regular opcode and the memory address are stored in the table, the new breakpoint opcode is stored in the instruction, and then the instruction is stored back into the memory location using the computer function 'storem().'

The breakpoint opcode is formed by taking the debug breakpoint code, which in this implementation is '0812', and adding the break number to the front of the code. For example, if the next open entry in the break table is at position 4, the break opcode computed would be '4812'. (Currently, up to eight (8) breakpoints may be set at any one time during the debug session.)

e. Set Program Counter

This operation gives the user the ability to set the program counter to any location in memory that contains an instruction value. Restricting the target memory location this way prevents a program execution error by the computer. In other words, trying to execute a non-instruction value causes the computer to generate an execution-terminating error. The command syntax is as follows:

'a' '*' segment:offset

It the user enters a memory location that does not contain an instruction, the operation self-aborts and the program counter remains unchanged.

3. Remove Breakpoint Operation

This operation undoes the 'set breakpoint' operation. Its command syntax is as follows:

'r' brknum

it removes breakpoints by their entry number in the break table. It first checks to see if the entry is in the table. If it is not, the operation self-aborts (and informs the user). If it is, the operation uses 'fetchm()' (with the break table entry memory address as a parameter) to retrieve the instruction value, inserts the regular opcode from the break table into the

instruction value, deletes the break table entry and stores the value back into memory using 'storem().' A sample demonstration of this operation can be found in Appendix A, Sample Sessions.

4. Trace Execution Operations

These operations are used to turn on and off the debug trace flag. They can also be used to trace a certain number of instructions and return control to the debugger. After the operation is set, it's is activated by issuing the 'go execute' command. Sample demonstrations can be found in Appendix A, Sample Sessions. The following subparagraphs present the functional details for each of the trace operations.

a. Trace On

This operation turns on the debug trace flag. When the flag is on, each instruction is displayed before it is executed. The command syntax is as follows:

't' '!'

The operation can be disabled at the debugger level by the 'trace off' command. The operation is also aborted during program execution by program termination or a breakpoint being encountered. This operation partially disables the 'trace *n*' operation and completely disables the 'trace off' operation.

b. Trace Off

This operation turns off the debug trace flag. When the flag is off, the instruction about to be executed is not displayed to the user. Its command syntax is as follows:

't' 'z'

This operation also disables the 'trace on' and 'trace *n*' commands at the debugger level. It can be disabled by the 'trace on' and 'trace *n*' commands at the debugger level.

c. Trace *n*

This operation causes execution to be traced for *n* instructions and then control to be transferred back to the debugger. Its command syntax is as follows:

't' span

While at the debugger command level, it can completely disable the 'trace off' operation. After the specified number of instructions are executed, the debug trace is again turned off. While at the debugger command level, this operation can be partially disabled by the 'trace on' command and totally disabled by the 'trace off' command. During program execution, the operation is disabled by program termination or a breakpoint being encountered. The number of instructions to be traced can also be overriden by the 'go execute *n*' command.

5. Go Operations

These commands are used to transfer control from the debugger back to the computer. Demonstrations of the operations can be found in Appendix A, Sample Sessions. The functional details of each of the go operations are presented in the following subparagraphs.

a. Go Uncontrolled

This operation transfers control back to the computer so that it can proceed with program execution. The operation proceeds until a

breakpoint is encountered, a 'trace *n*' operation is complete or the program terminates. Its command syntax is as follows:

'g' '!'

b. Go *n*

Like the 'go uncontrolled' operation, this operation causes control to be transferred from the debugger back to the computer for program execution. However, after *n* instructions are executed, control is transferred back to the debugger. Its command syntax is as follows:

'g' span

The operation is disabled in program execution by a breakpoint being encountered, program termination or the completion of the specified number of instructions. This operation has an indirect effect upon the 'trace *n*' operation. Changing the number of instructions to be executed in the 'go *n*' command also overrides the number to be traced.

6. Help Operation

This operation lists the available debugger operations and their command formats. Its has the following command syntax:

'?'

A demonstration of the operation is shown in Appendix A, Sample Sessions.

D. DEBUGGER CONTROL OF MACHINE EXECUTION

For the debugger to control the execution of the computer requires that certain "toggles" be added to the machine. Implementing these "toggles"

translates into defining control variables that the debugger sets and the computer reads to alter machine execution and flow of control. Figure 4.1 on page 37 is a list of the control variables added to this AM implementation. A more detailed description of the function of each of the control variables is presented in the following subparagraph.

1. Debugflag

This control variable directs the computer to activate and transfer control to the interactive debugger. It is set on two occasions, the user specifying the '-d' option at computer "startup" and the program terminating, normally or abnormally. It is reset when control is passed from the debugger back to the computer.

2. Debgtask

This control variable keeps the retrieve operations from generating an error if the debugger attempts to retrieve a value from an uninitialized storage location. If the storage location is uninitialized, the retrieve function generates a null value and returns it to the debugger. The termination of each debugger operation zeroes the variable.

3. Left2do

This variable controls the number of instructions to be executed for a 'go *n*' or a 'trace *n*' debugger operation. This control variable is zeroed if an execution error occurs or a breakpoint is encountered. It also works in tandem with the 'debgcntl' control variable.

4. Debgcntl

This control variable tells the computer that a 'go *n*' or a 'trace *n*' operation is being performed in tandem with program execution. If 'debgcntl'

```
int debugflag = 0;
    /* When == 1, calls the interactive debugger. */

int debgtask = 0;
    /* When == 1, tells AM that debugger is directly using AM functions for
    debugger operations. */

int left2do = 0;
    /* When == 1, tells AM how many instructions to do before forking back
    to the debugger. Set by 'go n' and 'trace n' debug ops. */

int debgcntl = 0;
    /* When == 1, tells AM that its execution is under control of a 'go n' or
    'trace    n' debug operation. */

int breakflg = 0;
    /* When == 1, tells AM that breakpoint encountered in its execution and
    conversionof an instruction must be made. */

int errorflg = 1;
    /* When == 0, tells error() that ICSTOP instruction has occurred and not
    to   print certain error messages. */

int dbgtrace = 0;
    /* When == 1, sends value at the _pc.val to standard output device, thus
    performing a program execution trace. */
```

Figure 4.1: Debugger Control Variables

is set, then the control variable 'left2do' is checked for equality with zero. This variable is zeroed when the number of instructions has been executed, a breakpoint occurs, or the program terminates.

5. Breakflg

Breakpoint opcodes are substituted for the regular instruction opcodes, with the regular opcode being stored in the break table. The 'breakflg' "toggle" is set when an instruction containing a break opcode is encountered. The variable signals the debugger that the instruction must be restored to an executable form by reinserting its regular opcode from the break table. 'Breakflg' is zeroed after the instruction is modified. When the debugger transfers control back to the computer, the modified instruction is then interpreted and executed.

6. Errorflg

In the AM's current implementation, normal as well as abnormal program termination calls the error handler to halt execution. This flag is set by the ICSTOP instruction so that certain error messages are not printed.

7. Dbgtrace

This "toggle" causes the computer to display the instruction to the user prior to its execution. It is set by the 'trace *n*' and 'trace on' operations. It is zeroed by a breakpoint being encountered, by the completion of the 'trace *n*' operation, by the 'trace off' operation, or program termination.

E. ERROR HANDLING IN PROGRAM EXECUTION

The control variables added to this AM implementation, coupled with the error handler modification, give the debugger the means for "trapping" control of program execution. Calling the error handling module now causes the debugging control variables to be zeroed, the appropriate messages to be displayed to the user, and a return to the debugger command level so that the

user can examine or alter the machine state and, if desired, rerun the program.

F. MODIFICATIONS TO AM IMPLEMENTATION

The main debugger program files added to this implementation of the AM are contained in Appendix C, Debugger Program Files. Some modifications were also made to the actual implementation of the AM. These changes were necessary to establish an interface between the debugger and the AM's physical resources. The more significant changes are briefly listed below:

- Debugger control variables were added to the AM implementation to provide proper transfer of control between the computer and the debugger;
- The memory, register and stack retrieve operations were modified to return a null value to the debugger if the storage location was uninitialized;
- A breakpoint opcode was added to the computer opcode definitions;
- A function for storing a value at a stacktop was added to the value retrieve and store module;
- The original copy-value function was duplicated and renamed for use in the AM assembler. This was done so as to hide the separate debugging process from program assembly process;
- The display-value function was modified to suppress displaying this AM's implementation details to the user;
- The error handler function was modified to call the debugger upon program termination, normal or abnormal.

V. CONCLUSIONS AND FUTURE WORK

Designing the interactive debugger to make use of the AM implementation's existing functions again demonstrates the advantage of formally specifying functional interfaces for computer resources. Because the interfaces were well defined and built as conceptual "black boxes," linking them to the debugger was relatively straightforward. Having the interfaces being built as "black boxes" also helped to prevent the modification "ripple effect" upon the behavior of existing functions. It is expected that the addition of an interactive debugger to the AM programming environment will significantly aid future developers of AM resource tools.

Although the interactive debugger significantly enhances the AM programmers ability to interact with the other elements in the AM programming environment, its interface to the user can be improved. In light of this, the following areas for continuing research are suggested:

- Implement an in-line assembler for the debugger. This would permit the user to set storage resources by entering the actual assembler language statements. This provides a debugger interface at the level of the assembly language programming abstraction;
- Implement an in-line disassembler for displaying instruction values in memory;
- Implement a graphical user interface to the debugger.

APPENDIX A: SAMPLE SESSIONS

```
>am -d

        ******************************
        *      THE DEBUGGER          *
        ******************************

                    HELP OPERATION

Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)

>?
          Debugger Commands
        ************************
'd'(isplay)     'm'(emory),   {'*' | seg:offset}, span
                'r'(egister),   seg:offset, span
                's'(tack - top only),   seg
                'b'(reaks - all)
                '*'(program counter)
'g'(o)          {'!' | n <instrs>}
'?'(list available debug commands)
's'(et)         'm'(emory), seg:offset, val_type, val
                'r'(egister), seg:offset, val_type, val
                's'(tacktop, seg, val_type val
                'b'(reak),  seg:offset
                '*'(program counter), seg:offset
't'(race)       '!'(on)       < TRACE
                'z'(off)                STARTED
                for n instrs                    BY 'GO'>
'q'(uit debug and halt execution)
Legend: | - or, [] - optional, <> - comment, {} - Must choose an item.

Enter non-blank char to continue.
>q
                  DISPLAY OPERATIONS

                    Display Memory
Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
```

```
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>m

Enter one of following:
        'v' - addr value prompt
        '*' - for current PC value
        '@' - to abort the operation:
>*
        * OPERATION SPAN *

Enter decimal number between 1 and 20
        or
'@' to abort the operation:
>16

Memaddr Contents

00000000 (V_INSTR) ISPSHI_
                (V_FILE) 2
                (V_SAD) (0 : 0)
00000001 (V_INSTR) ISPSHI_
                (V_MAD) (1 : 0)
                (V_SAD) (0 : 0)
00000002 (V_INSTR) IFWRITE
                (V_SAD) (0 : 0)
00000003 (V_INSTR) ISPSHI_
                (V_FILE) 2
                (V_SAD) (0 : 0)
00000004 (V_INSTR) ISPSHI_
                (V_MAD) (1 : 1)
                (V_SAD) (0 : 0)
00000005 (V_INSTR) IFWRITE
                (V_SAD) (0 : 0)
00000006 (V_INSTR) ISPSHI_
                (V_FILE) 2
                (V_SAD) (0 : 0)
00000007 (V_INSTR) ISPSHI_
                (V_MAD) (1 : 0)
                (V_SAD) (0 : 0)
00000008 (V_INSTR) IFWRITE
```

```
                (V_SAD)  (0 : 0)
00000009 (V_INSTR)  ISPSHI_
                (V_FILE)  2
                (V_SAD)  (0 : 0)
0000000A (V_INSTR)  ISPSHI_
                (V_MAD)  (1 : 1)
                (V_SAD)  (0 : 0)
0000000B (V_INSTR)  IFWRITE
                (V_SAD)  (0 : 0)
0000000C (V_INSTR)  ICSTOP
0000000D (V_NULL) 0
0000000E (V_NULL) 0
0000000F (V_NULL) 0
```

Display Register

```
Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>r

Enter one of following:
        'v' -  addr value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 0
        or
'@' to abort the operation:
>0

Enter decimal offset between 0  and 31
        or
'@' to abort the operation:
>0

        * OPERATION SPAN *
```

```
Enter decimal number between 1  and 20
        or
'@' to abort the operation:
>5

Regnum  Contents

00000000 (V_NULL) 0
00000001 (V_NULL) 0
00000002 (V_NULL) 0
00000003 (V_NULL) 0
00000004 (V_NULL) 0
```

Display Stacktop

```
Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>s

Enter one of following:
        'v' -  segment value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 0
        or
'@' to abort the operation:
>0

Top of Stack:
000001FF (V_NULL) 0
```

Display Breakpoints

```
Enter letter of operation:

        d (isplay)
        g (o execute)
```

```
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>b

Enter '!' to continue
        or
'@' to abort the operation:
>!

        ************************
        *                      *
        *     BREAKPOINTS      *
        *                      *
        ************************

BRKNUM   MEMADDR   OPCODE
   0
   1
   2
   3
   4
   5
   6
   7
```

Display Program Counter

```
Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
```

```
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>*

Enter one of following:
        '!' to confirm display pc
        '@' to abort the operation
>!

        PRGM COUNTER in segment 0 at offset 0.
```

SET OPERATIONS

Set Memory

```
Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>s

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>m

Enter one of following:
        'v' -  memaddr value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 1
        or
'@' to abort the operation:
>0

Enter decimal offset between 0  and 1023
        or
'@' to abort the operation:
>13
```

```
          **** Entering Value to be Stored ****

Enter number besides type desired:

     1 - BOOL    2 - NAT          3 - INT        4 - CHAR
     5 - CSTR    6 - MAD          7 - RAD        8 - SAD
     9 - FILE    10 - INSTR       11 - MOP       12 - DOP
    13 - ROP     14 - BOP         @ - abort op
>10

Enter HEX opcode
          or
'@' to abort the operation:
>3831

          ***    Entering Operand #1    ***
          **** Entering Value to be Stored ****

Enter number besides type desired:

     1 - BOOL    2 - NAT          3 - INT        4 - CHAR
     5 - CSTR    6 - MAD          7 - RAD        8 - SAD
     9 - FILE    11 - MOP         12 - DOP       13 - ROP
    14 - BOP     @ - abort op
>3

Enter decimal number between

          -2147483647 & 2147483647 (no ','):
or
          '@' to abort the operation:
>500

          ***    Entering Operand #2    ***
          **** Entering Value to be Stored ****

Enter number besides type desired:

     1 - BOOL    2 - NAT          3 - INT        4 - CHAR
     5 - CSTR    6 - MAD          7 - RAD        8 - SAD
     9 - FILE    11 - MOP         12 - DOP       13 - ROP
    14 - BOP     @ - abort op
>7

Enter one of following:
          'v' -  regaddr value prompt
          '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 0
          or
```

```
'@' to abort the operation:
>0

Enter decimal offset between 0  and 31
        or
'@' to abort the operation:
>31

Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>m

Enter one of following:
        'v' -  addr value prompt
        '*' -  for current PC value
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 1
        or
'@' to abort the operation:
>0

Enter decimal offset between 0  and 1023
        or
'@' to abort the operation:
>13

        * OPERATION SPAN *

Enter decimal number between 1  and 20
        or
'@' to abort the operation:
>2

Memaddr  Contents
```

```
0000000D (V_INSTR)  IM_I_R_
                (V_INT)  500
                (V_RAD)  (0 : 31)
0000000E (V_NULL) 0
```

Set Register

```
Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>s

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>r

Enter one of following:
        'v' -  regaddr value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 0
        or
'@' to abort the operation:
>0

Enter decimal offset between 0  and 31
        or
'@' to abort the operation:
>13

        **** Entering Value to be Stored ****

Enter number besides type desired:

    1 - BOOL    2 - NAT          3 - INT        4 - CHAR
    5 - CSTR    6 - MAD          7 - RAD        8 - SAD
    9 - FILE    10 - INSTR       11 - MOP       12 - DOP
   13 - ROP     14 - BOP         @ - abort op
>8
```

```
Enter one of following:
        'v' -  stkaddr value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 0
        or
'@' to abort the operation:
>0

Enter decimal offset between 0  and 511
        or
'@' to abort the operation:
>56

Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>r

Enter one of following:
        'v' -  addr value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 0
        or
'@' to abort the operation:
>0

Enter decimal offset between 0  and 31
        or
'@' to abort the operation:
>13

        * OPERATION SPAN *
```

```
Enter decimal number between 1  and 20
        or
'@' to abort the operation:
>2

Regnum  Contents

0000000D (V_SAD)  (0 : 56)
0000000E (V_NULL) 0
```

Set Stacktop

```
Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>s

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>s

Enter one of following:
        'v' -  regaddr value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment * between 0  and 0
        or
'@' to abort the operation:
>0

        **** Entering Value to be Stored ****

Enter number besides type desired:

    1 - BOOL    2 - NAT          3 - INT        4 - CHAR
    5 - CSTR    6 - MAD          7 - RAD        8 - SAD
    9 - FILE    10 - INSTR       11 - MOP       12 - DOP
   13 - ROP     14 - BOP         @ - abort op
>4

Enter character
```

```
        or
'@' to abort the operation:
>!

Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>s

Enter one of following:
        'v' -  segment value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 0
        or
'@' to abort the operation:
>0

Top of Stack:
00000 1FF (V_CHAR) !
```

Set Breakpoints

```
Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>s

Enter one of choices below:

        m (emory)
```

```
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>b

Enter one of following:
        'v' -  memaddr value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 1
        or
'@' to abort the operation:
>0

Enter decimal offset between 0  and 1023
        or
'@' to abort the operation:
>2

Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>s

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>b

Enter one of following:
        'v' -  memaddr value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 1
        or
'@' to abort the operation:
>0

Enter decimal offset between 0  and 1023
```

```
        or
'@' to abort the operation:
>4

Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>s

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>b
Enter one of following:
        'v' -  memaddr value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 1
        or
'@' to abort the operation:
>0

Enter decimal offset between 0  and 1023
        or
'@' to abort the operation:
>9
Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>s

Enter one of choices below:

        m (emory)
        r (egister)
```

```
        s (tack top)
        b (reak)
        * (program counter)
>b

Enter one of following:
        'v' -  memaddr value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 1
        or
'@' to abort the operation:
>0

Enter decimal offset between 0  and 1023
        or
'@' to abort the operation:
>11

Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>s

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>b

Enter one of following:
        'v' -  memaddr value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 1
        or
'@' to abort the operation:
>0

Enter decimal offset between 0  and 1023
        or
```

```
'@' to abort the operation:
>13

Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>s

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>b

Enter one of following:
        'v' -  memaddr value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 1
        or
'@' to abort the operation:
>0

Enter decimal offset between 0  and 1023
        or
'@' to abort the operation:
>12

Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
```

```
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>b

Enter '!' to continue
        or
'@' to abort the operation:
>!

        ************************
        *                      *
        *    BREAKPOINTS       *
        *                      *
        ************************

BRKNUM   MEMADDR   OPCODE
   0
   1
   2   0000000C   ICSTOP
   3   0000000D   IM_I_R_
   4   0000000B   IFWRITE
   5   00000009   ISPSHI_
   6   00000004   ISPSHI_
   7   00000002   IFWRITE
```

Set Program Counter

```
Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>s

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>*

Enter one of following:
        'v' - prog cntr value prompt
        '@' - to abort the operation:
>v
```

```
Enter decimal segment # between 0  and 1
        or
'@' to abort the operation:
>0

Enter decimal offset between 0  and 1023
        or
'@' to abort the operation:
>16

Sorry, non-instr at memaddr.
Program counter unchanged.

Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>s

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>*

Enter one of following:
        'v' -  prog cntr value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 1
        or
'@' to abort the operation:
>0

Enter decimal offset between 0  and 1023
        or
'@' to abort the operation:
>2

Enter letter of operation:

        d (isplay)
```

```
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>*

Enter one of following:
        '!' to confirm display pc
        '@' to abort the operation
>!

        PRGM COUNTER in segment 0 at offset 2.


Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>s

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>*

Enter one of following:
        'v' -  prog cntr value prompt
        '@' -  to abort the operation:
>v

Enter decimal segment # between 0  and 1
        or
```

'@' to abort the operation:
>0

Enter decimal offset between 0 and 1023
or
'@' to abort the operation:
>0

TRACE / GO OPERATIONS

Trace On with Go Uncontrolled

Enter letter of operation:

d (isplay)
g (o execute)
? (list debug ops)
r (emove break)
s (et)
t (race)
q (uit & halt exec)
>t

Enter one of following:
Decimal number between 1 and 20
'!' for 'trace on'
'z' for 'trace off'
'@' to abort the operation:
>!

Enter letter of operation:

d (isplay)
g (o execute)
? (list debug ops)
r (emove break)
s (et)
t (race)
q (uit & halt exec)
>g

Enter one of choices below:
Decimal range btwn 1 and 20
'!' - uncontrolled go
'@' - to abort the operation
>!

```
00000000 (V_INSTR)  ISPSHI_
                (V_FILE)  2
                (V_SAD)  (0 : 0)
00000001 (V_INSTR)  ISPSHI_
                (V_MAD)  (1 : 0)
                (V_SAD)  (0 : 0)
```

```
        ** BREAKPOINT ENCOUNTERED **

        ******************************
        *      THE DEBUGGER          *
        ******************************
```

Trace Off with Go Uncontrolled

```
Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>t

Enter one of following:
        Decimal number between 1 and 20
        '!' for 'trace on'
        'z' for 'trace off'
        '@' to abort the operation:
>z

Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>g

Enter one of choices below:
        Decimal range btwn 1 and 20
        '!' - uncontrolled go
        '@' - to abort the operation
>!

        ** BREAKPOINT ENCOUNTERED **

        ******************************
        *      THE DEBUGGER          *
        ******************************


Enter letter of operation:
```

```
        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>*

Enter one of following:
        '!' to confirm display pc
        '@' to abort the operation
>!

        PRGM COUNTER in segment 0 at offset 4.
```

Go *n* Instructions

```
Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>g

Enter one of choices below:
        Decimal range btwn 1 and 20
        '!' - uncontrolled go
        '@' - to abort the operation
>2

Indicated number of instrs done.
        ******************************
        *      THE DEBUGGER          *
        ******************************

Enter letter of operation:
```

```
        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>*

Enter one of following:
        '!' to confirm display pc
        '@' to abort the operation
>!

        PRGM COUNTER in segment 0 at offset 6.


Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>b

Enter '!' to continue
        or
'@' to abort the operation:
>!
```

```
        ************************
        *                      *
        *     BREAKPOINTS      *
        *                      *
        ************************

BRKNUM   MEMADDR   OPCODE
   0
   1
   2   0000000C   ICSTOP
   3   0000000D   IM_I_R_
   4   0000000B   IFWRITE
   5   00000009   ISPSHI_
   6   00000004   ISPSHI_
   7   00000002   IFWRITE
```

Remove Breakpoints

```
Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>r

Enter decimal break number between 0 and 7
        or
'@' to abort the operation:
>0

Breakpoint not in Table.

Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>r
Enter decimal break number between 0 and 7
        or
'@' to abort the operation:
>7

Enter letter of operation:
```

```
        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>b

Enter 'I' to continue
        or
'@' to abort the operation:
>!

        *************************
        *                       *
        *      BREAKPOINTS      *
        *                       *
        *************************

BRKNUM   MEMADDR   OPCODE
   0
   1
   2   0000000C   ICSTOP
   3   0000000D   IM_I_R_
   4   0000000B   IFWRITE
   5   00000009   ISPSHI_
   6   00000004   ISPSHI_
   7
```

Stepping Through Program Execution

```
Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>g
```

```
Enter one of choices below:
        Decimal range btwn 1 and 20
        '!' - uncontrolled go
        '@' - to abort the operation
>!

        ** BREAKPOINT ENCOUNTERED **

        ******************************
        *      THE DEBUGGER            *
        ******************************


Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>*

Enter one of following:
        '!' to confirm display pc
        '@' to abort the operation
>!

        PRGM COUNTER in segment 0 at offset 9.


Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>r
```

```
Enter decimal break number between 0 and 7
        or
'@' to abort the operation:
>4

Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>g

Enter one of choices below:
        Decimal range btwn 1 and 20
        '!' - uncontrolled go
        '@' - to abort the operation
>!

        ** BREAKPOINT ENCOUNTERED **

        ******************************
        *      THE DEBUGGER          *
        ******************************


Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>d

Enter one of choices below:

        m (emory)
        r (egister)
        s (tack top)
        b (reak)
        * (program counter)
>*

Enter one of following:
        '!' to confirm display pc
        '@' to abort the operation
```

```
>!

        PRGM COUNTER in segment 0 at offset 12.


Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>g

Enter one of choices below:
        Decimal range btwn 1 and 20
        '!' - uncontrolled go
        '@' - to abort the operation
>!

AM: End of execution 32
   _pc=0000000C
        ********************************
        *       THE DEBUGGER           *
        ********************************


Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>q

Enter one of following:
        '!' to END DEBUG AND EXECUTION
        '@' to abort operation:
>@

Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
```

```
        t (race)
        q (uit & halt exec)
>g

Enter one of choices below:
        Decimal range btwn 1 and 20
        '!' - uncontrolled go
        '@' - to abort the operation
>@


Enter letter of operation:

        d (isplay)
        g (o execute)
        ? (list debug ops)
        r (emove break)
        s (et)
        t (race)
        q (uit & halt exec)
>q

Enter one of following:
        '!' to END DEBUG AND EXECUTION
        '@' to abort operation:
>!

Exiting Debugger, Halting Execution.
```

APPENDIX B: DEBUGGER COMMAND SYNTAX SYMBOLS

The following symbols are used to describe the general syntax of all the debugger comands:

'd'	-	DISPLAY
's'	-	SET or STACK (depending on context)
'g'	-	GO AND EXECUTE
'?'	-	LIST AVAILABLE DEBUG COMMANDS
'r'	-	REMOVE BREAKPOINT
't'	-	TRACE EXECUTION
'm'	-	MEMORY
'r'	-	REGISTER
'*'	-	PROGRAM COUNTER
segment	-	RESOURCE SEGMENT NUMBER
offset	-	SEGMENT OFFSET ADDRESS
span	-	NUMBER OF OPERATIONS TO BE PERFORMED
{}	-	ONE ITEM IN BRACES MUST BE CHOSEN
|	-	OR
'!'	-	ON or OK (depending on context)
'z'	-	OFF
brknum	-	BREAKPOINT NUMBER
value	-	DATA VALUE
val_type	-	DATA TYPE

APPENDIX C: DEBUGGER PROGRAM FILES

Debugger Header File

```
/* DEBUG.H : Basic typedefs, defines, and globals for the AM debugger.

     -AM version 1.0  - Z100
     -This file is included in all the debugger modules.

Changes:

*/

#ifndef     DEBUG_H
#define     DEBUG_H

/* display defines */

#define     DISPLMEM        000
#define     DISPLREG        010
#define     DISPLSTK        020
#define     DISPLBRK        030
#define     DISPL_PC        040

/* go define */

#define     GOEXEC          100

/* help define */

#define     HELP            200

/* remove breakpoint define */

#define     REMOVBRK        300

/* set defines */

#define     SETMEMR         400
#define     SETREGR         410
#define     SETSTK          420
#define     SETBRK          430
#define     SET_PC          440

/* trace defines */

#define     TRACEOP         500

/* quit defines */
```

```
#define             QUITDEBG        600

/* atomic types and defines */

#define             UNDEFND         -1
#define             TRUE            1
#define             FALSE           0
#define             MAXBRKS         8       /* Max # of breaks allowed
*/
#define             MAXEXECS        20      /* Max # instrs to execute
                                            before returning to debug */

#define             MAXLINES        20      /* Max items on debug screen
*/

#define             MXDECSTR        12      /* Max chars in DECIMAL input
                                            string; 10 for chars in INT
                                            string, 1 for null string
                                             terminator and 1 for sign */

#define             MXHEXSTR        5       /* Max chars in HEXIDECIMAL
input
                                            string; 4 for chars in INT
                                            string and 1 for null string
                                            terminator. */

#define             MXINPSTR        2       /* Max chars in PROMPT input
                                            string; 1 for char in prompt
                                            string and 1 for null string
                                            terminator. */

#define             TRACEON         -1      /* Couldn't use 1 due to span
                                            conflict.*/
#define             TRACEOFF        0

typedef     int     BOOLN;  /* So named to prevent AM def conflict */

typedef     struct  {
    int     oprtn;
    long    rngebegn;
    long    rngespan;
    VAL     *val;               /* VAL typedef in amtype.h */
    BOOLN   abortop;
} OPTION;

typedef             struct  {
    address memaddr;
    short   opcdval;
} BREAKS;

#endif
```

Debugger Driver File

```
/* DEBUG.C : Driver module for the AM debugger.
     -AM version 1.0  - Z100
Changes:

*/

#include "amdef.h"
#include "amtype.h"
#include "amextern.h"
#include <setjmp.h>
#include "debug.h"

/* EXTERNAL REFERENCES */

extern      jmp_buf  *_context;          /* defined in main()/am.c */

extern      char     *stripblk();        /* from debugutl.c */

extern      short    getopnd();          /* from aminstr.c */

extern      short    gtopcdex();         /* all from debugopr.c */
extern               displmem();
extern               displreg();
extern               displstk();
extern               displbrk();
extern               displ_pc();
extern               goexec();
extern               help();
extern               removbrk();
extern               setmemr();
extern               setregr();
extern               setstk();
extern               setbrk();
extern               set_pc();
extern               traceop();
extern               quitdebg();

/* GLOBAL VARIABLES */

OPTION      debg_opt;

/*GETRESRC()
     function:
               -This function prompts the user for the resource to be
               operated upon and returns the key character for the
               resource indicated.

     interface:
```

```
        called by:

                getopr()

        calls:
                stripblk()/debugutl.c

        errors:

*/

char
getresrc()

{       char *strptr;
        char inpstr[MXINPSTR];      /* MXINPSTR = 2 */

        do {
                fprintf(stdout,"Enter  one of choices below:\n\n");
                fprintf(stdout,"\t\tm  (emory)\n\t\tr (egister)\n");
                fprintf(stdout,"\t\ts  (tack top)\n\t\tb (reak)\n");
                fprintf(stdout,"\t\t*  (program counter)\n\n");

                fscanf(stdin,"%ls",inpstr);

                strptr = stripblk(inpstr);

                if (strlen(strptr) == 0) {
                        fprintf(stdout,"No  choice entered.\n");
                        fprintf(stdout,"One  MUST be specified.\n\n");
                }
                else
                        switch(strptr[0]) {
                                case 'm': return('m');

                                case 'r':  return('r');

                                case 's':  return('s');

                                case 'b':  return('b');

                                case '*':  return('*');

                                default:

                                 fprintf(stdout,"Invalid  response.\n");

                        }

        }
        while (TRUE);
```

```
}

/*GETOPR()
     function:
          -This function maps the ascii user operation request
          into an debugger operation code and returns the code.

     interface:

     called by:
          debug()

     calls:
          stripblk()/debuguti.c
          getresrc()

     errors:

*/

int
getopr()

{    char *strptr;
     char inpstr[MXINPSTR];      /* MXINPSTR = 2 */

     do {
          fprintf(stdout,"\nEnter  letter  of operation:\n\n");
          fprintf(stdout,"\t\td  (isplay)\n\t\tg  (o execute)\n");
          fprintf(stdout,"\t\t?  (list debug ops)\n");
          fprintf(stdout,"\t\tr  (emove break)\n\t\ts (et)\n");
          fprintf(stdout,"\t\tt  (race)\n\t\tq (uit & halt exec)\n\n");
          fscanf(stdin,"%ls",inpstr);

          strptr = stripblk(inpstr);

          if (strlen(strptr) == 0) {
               fprintf(stdout,"No  operation entered.\n");
               fprintf(stdout,"One  MUST be specified.\n\n");
          }
          else
           switch(strptr[0]) {
               case 'd':
               case 's':  switch(getresrc()) {
                              case 'm': if (strptr[0] == 'd')
                                            return(DISPLMEM);
                                        return(SETMEMR);

                              case 'r': if (strptr[0] == 'd')
                                            return(DISPLREG);
                                        return(SETREGR);
```

```
                                case 's': if (strptr[0] == 'd')
                                            return(DISPLSTK);
                                          return(SETSTK);

                                case 'b': if (strptr[0] == 'd')
                                            return(DISPLBRK);
                                          return(SETBRK);
                                case '*': if (strptr[0] == 'd')
                                            return(DISPL_PC);
                                          return(SET_PC);

                                }

                        case 'g': return(GOEXEC);

                        case '?': return(HELP);

                        case 'r': return(REMOVBRK);

                        case 't': return(TRACEOP);

                        case 'q': return(QUITDEBG);


                        default:  fprintf(stdout,"Invalid  operation.\n");

                }

        }
        while (TRUE);

}

/*DEBUG()
        function:
                -This is the driver function for the interactive debug-
                ger.

        interface:
                (x) breakflg/am.h
                (p) i                   instruction pointer
                (p) m                   program counter value
                (g) debg_opt            variable for debugger operation

        called by:
                main()/am.c

        calls:
                getopnd()/aminstr.c
                gtopcdex()/debugopr.c
                getopr()/
```

```
            displmem()/debugopr.c
            displreg()/debugopr.c
            displstk()/debugopr.c
            displbrk()/debugopr.c
            displ_pc()/debugopr.c
            goexec()/debugopr.c
            help()/debugopr.c
            removbrk()/debugopr.c
            setmemr()/debugopr.c
            setregr()/debugopr.c
            setstk()/debugopr.c
            setbrk()/debugopr.c
            set_pc()/debugopr.c
            traceop()/debugopr.c
            quitdebg()/debugopr.c


    errors:

*/

debug(i,m)
INSTR *i;
MAD *m;

{   short   brknum;
    OPTION *opt;
    BOOLN diff_pc = FALSE;

    fprintf(stdout,"\t\t*******************************\n");
    fprintf(stdout,"\t\t*        THE DEBUGGER                 *\n");
    fprintf(stdout,"\t\t*******************************\n\n");

    /* Exchange the debug opcodefor the regular opcode*/

    if (breakflg) {
            brknum = getopnd(i->val[0].opcdval);
            i->val[0].opcdval = gtopcdex(brknum);
            breakflg = 0;
    }

    opt = &debg_opt;

    do {
            opt->oprtn = UNDEFND;
            opt->rngebegn =  UNDEFND;
            opt->rngespan =  UNDEFND;
            opt->val =  NULL;
            opt->abortop = FALSE;

            switch(getopr()) {
```

```
                case DISPLMEM:      displmem(opt);
                                    break;

                case DISPLREG:      displreg(opt);
                                    break;

                case DISPLSTK:      displstk(opt);
                                    break;

                case DISPLBRK:      displbrk(opt);
                                    break;

                case DISPL_PC:      displ_pc(opt);
                                    break;

                case GOEXEC:        goexec(opt);
                                    break;

                case HELP:          help(opt);
                                    break;

                case REMOVBRK:      removbrk(opt);
                                    break;

                case SETMEMR:       setmemr(opt);
                                    break;

                case SETREGR:       setregr(opt);
                                    break;

                case SETSTK:        setstk(opt);
                                    break;

                case SETBRK:        setbrk(opt);
                                    break;

                case SET_PC:        set_pc(opt);
                                    break;

                case TRACEOP:       traceop(opt);
                                    break;

                case QUITDEBG:      quitdebg(opt);

        }

        if ((!(opt->abortop)) && (opt->oprtn == GOEXEC)) {

                if (diff_pc) {
                        debugflag = 0;      /* Keeps debugger from
                                               being called after setjmp()
                                               in main()/am.c */
```

```
                        longjmp(_context,1);
                    }

                    return;
                }

                if ((!(opt->abortop)) && (opt->oprtn == SET_PC))
                    diff_pc = TRUE;

        }
        while (TRUE);
}
```

Debugger Operations File

```
/* DEBUGOPR.C : This module contains the SET initialization  functions
                for the AM debugger.

     -AM version 1.0  - Z100

Changes:

*/

#include  "amdef.h"
#include  "amtype.h"
#include  "amextern.h"
#include  "debug.h"

/* EXTERNAL FUNCTIONS */

extern      char      *stripblk();        /* both from debugutl.c  */
extern                str2hex();
extern                str2dec();
extern      address   cnv2addr();

extern      short     getopcode();        /* both from aminstr.c  */
extern      short     getopnd();

extern      char      *pmalloc();         /* all from amstate.c */
extern                fmalloc();
extern      VAL       *fetchm();
extern      STATE     storem();
extern      VAL       *fetchr();
extern      STATE     storer();
extern      VAL       *topstk();

extern                storestk();         /* an EXCLUSIVE debugger function
in
                                          in amstate.c */

extern      char      *amdefs();          /* from amcutl.c */

/* LOCAL GLOBAL VARIABLES */

BOOLN       inst_get  = FALSE; /* When true, used to keep getvalue() from
                                  allowing recursive calls to get_inst() */
long _segnum;
long _offset;

static      BREAKS    brktable[MAXBRKS];
static      short     mt_slots[MAXBRKS];
static      short     topslot;
```

```
static      BOOLN   mt_init = FALSE;
                    /* Tells if mt_slots[] initialized.  Note the
                     initialization  is done once!(ref 'C' manual)*/


/*GTOPCDEX()
     function:
          -returns the opcodestored in the brktable at slot 'brknum'.

     interface:
          (p) brknum
          (g) brktable[]

     called by:
          debug()/debug.c
          copyval()/amcutl.c
          showmem()/amcutl.c

     calls:

     errors:


*/

gtopcdex(brknum)
short       brknum;

{
     if (brktable[brknum].opcdval == UNDEFND)
          return(0);

     return(brktable[brknum].opcdval);
}

/*GETRANGE()
     function:
          -This function prompts the user for range value.

     interface:
          (p) opt
          (p) maxnum

     called by:
          dipslmem()
          displreg()
          str2dec()/debugutl.c

     calls:

     errors:
```

```
*/

getrange(opt,maxnum)
OPTION *opt;
int maxnum;

{       char *strptr;
        char inpstr[MXDECSTR];
        long number;
        BOOLN validnum;

        do {
                fprintf(stdout,"\t\t*   OPERATIONSPAN *\n\n");
                fprintf(stdout,"Enter  decimal number between 1  ");
                fprintf(stdout,"  and %d\n\t\tor\n",maxnum);
                fprintf(stdout,"'@'  to abort the operation: \n");
                fscanf(stdin,"%10s",inpstr);

                strptr  = stripblk(inpstr);

                if (strptr[0] == '@') {
                        opt->abortop = TRUE;
                        return;
                }

                str2dec(strptr,&validnum,&number);
                if (!validnum) {
                        fprintf(stdout,"Invalid  number entered.\n");
                        continue;
                }

                if ((number < 0) || (number > maxnum)) {
                        fprintf(stdout,"Number  out of range.\n");
                        continue;
                }

                opt->rngespan = number;
                return;
        }
        while (TRUE);

}

/*GET_SEGM()
        function:
                -This function prompts the user for segment value.

        interface:
                (p) mxsegnum
                (p) abortop

        called by:
```

```
                dipslmem()
                displreg()
                displstk()
                setmemr()
                setregr()
                setstk()
                setbrk()
                set_pc()
                get_rad()
                get_sad()
                get_mad()

        calls:
                stripblk()/debugutl.c
                str2dec()/debugutl.c

        errors:

*/

long
get_segm(abortop, mxsegnum)
BOOLN *abortop;
int mxsegnum;

{       char *strptr;
        char inpstr[MXDECSTR];
        long number;
        BOOLN validnum;

        do {
                fprintf(stdout,"Enter  decimal segment # between 0 ");
                fprintf(stdout,"  and %d\n\t\tor\n",mxsegnum);
                fprintf(stdout,"'@'  to abort the operation: \n");
                fscanf(stdin,"%10s",inpstr);

                strptr = stripblk(inpstr);

                if (strptr[0] == '@') {
                        *abortop = TRUE;
                        return(0);
                }

                str2dec(strptr,&validnum,&number);
                if (!validnum) {
                        fprintf(stdout,"Invalid  number entered.\n");
                        continue;
                }

                if ((number >= 0) && (number <= mxsegnum)) {
                        return(number);
                }
```

```
                fprintf(stdout,"Segment  # out of range.\n");
                continue;

        }
        while (TRUE);

}

/*GET_OFST()
        function:
                -This function prompts the user for offset value.

        interface:
                (p) mxoffset
                (p) abortop

        called by:
                dipsimem()
                dispireg()
                setmemr()
                setregr()
                setbrk()
                set_pc()
                get_rad()
                get_sad()
                get_mad()

        calls:
                stripblk()/debugutl.c
                str2dec()/debugutl.c

        errors:

*/

long
get_ofst(abortop, mxoffset)
BOOLN *abortop;
long mxoffset;

{       char *strptr;
        char inpstr[MXDECSTR];
        long number;
        BOOLN validnum;

        do {
                fprintf(stdout,"Enter  decimal offset between 0 ");
                fprintf(stdout,"  and %d\n\t\tor\n",mxoffset);
                fprintf(stdout,"'@'  to abort the operation: \n");
                fscanf(stdin,"%10s",inpstr);
```

```
            strptr = stripblk(inpstr);

            if (strptr[0] == '@') {
                    *abortop = TRUE;
                    return(0);
            }

            str2dec(strptr,&validnum,&number);
            if (!validnum) {
                    fprintf(stdout,"Invalid  number entered.\n");
                    continue;
            }

            if ((number < 0) || (number > mxoffset)) {
                    fprintf(stdout,"Offset  out of range.\n");
                    continue;
            }

            return(number);
    }
    while (TRUE);

}

/*DISPLMEM()
    function:
            -This function performs the 'display memory' operation.

    interface:
            (p) opt
            (x) _numusrseg/am.h
            (x) _pc/am.h
            (x) _mem[]/am.h
            (g) _segnum
            (g) _offset
            (x) debgtask/am.h

    called by:
            debug()/debug.c

    calls:
            getrange()
            fetchm()/amstate.c
            showmem()/amcutl.c
            cnv2addr()/debugutl.c
            stripblk()/debugutl.c
            get_segm()
            get_ofst()

    errors:

*/
```

```
displmem(opt)
OPTION *opt;

{       char *strptr;
        char inpstr[MXINPSTR];
        long i;
        long number;
        long totalmem = 0;
        BOOLN validnum;
        MAD tmpaddr;

        opt->oprtn = DISPLMEM;

        do {
                fprintf(stdout,"Enter  one of following:\n");
                fprintf(stdout,"\t\t'v'  -  addr value prompt\n");
                fprintf(stdout,"\t\t'*'  -  for current PC value\n");
                fprintf(stdout,"\t\t'@'  -  to abort the operation: \n");
                fscanf(stdin,"%ls",inpstr);

                strptr = stripblk(inpstr);

                if (strptr[0] == '@') {
                        opt->abortop = TRUE;
                        return;
                }

                if (strptr[0] == '*') {
                        _segnum = (_pc.val & X_SEGMSK) >> X_SEGSFT;
                        _offset = _pc.val & X_ADRMSK;
                        getrange(opt,MAXLINES);
                        break;
                }

                if (strptr[0] == 'v') {
                        _segnum = get_segm(&opt->abortop, _numusrseg - 1);

                        if (opt->abortop)
                                return;

                        _offset = get_ofst(&opt->abortop,
                                                _mem[_segnum].size - 1);

                        if (opt->abortop)
                                return;

                        getrange(opt,MAXLINES);

                        break;
                }
```

```
                fprintf(stdout,"Incorrect response.\n");

        }
        while (TRUE);

        if (opt->abortop)
                return;

        debgtask = 1;

        fprintf(stdout,"Memaddr   Contents\n\n");

        if (opt->rngespan == UNDEFND) {
                tmpaddr.val = cnv2addr(_segnum_offset);
                showmem(&tmpaddr,fetchm(&tmpaddr,Q));
        }
        else {
                i = 0;

                while((i < opt->rngespan) && (_segnum < _numusrseg)) {
                  tmpaddr.val = cnv2addr(_segnum_offset);
                  showmem(&tmpaddr,fetchm(&tmpaddr,Q));
                  i++;
                  _offset++;
                  if (_offset == _mem[_segnum].size) {
                          _offset = 0;
                          _segnum++;
                  }
                }

        }

        debgtask = 0;

}

/*DISPLREG()
        function:
                -This function performs the 'display register' operation.

        interface:
                (p) opt
                (x) _numregseg/am.h
                (x) _reg[]/am.h
                (g) _segnum
                (g) _offset
                (x) debgtask/am.h

        called by:
                debug()/debug.c

        calls:
```

```
                getrange()
                fetchr()/amstate.c
                showmem()/amcutl.c
                cnv2addr()/debugutl.c
                stripblk()/debugutl.c
                get_segm()
                get_ofst()

        errors:

*/

displreg(opt)
OPTION *opt;

{       char *strptr;
        char inpstr[MXDECSTR];
        long i;
        long number;
        long numregs = 0;
        BOOLN validnum;
        RAD tmpreg;

        opt->oprtn = DISPLREG;

        do {
                fprintf(stdout,"Enter  one of following:\n");
                fprintf(stdout,"\t\t'v'  -  addr value prompt\n");
                fprintf(stdout,"\t\t'@'  -  to abort the operation: \n");
                fscanf(stdin,"%1s",inpstr);

                strptr = stripblk(inpstr);

                if (strptr[0] == '@') {
                        opt->abortop = TRUE;
                        return;
                }

                if (strptr[0] == 'v') {
                        _segnum = get_segm(&opt->abortop, _numregseg - 1);

                        if (opt->abortop)
                                return;

                        _offset = get_ofst(&opt->abortop,_reg[_segnum].num
- 1);

                        if (opt->abortop)
                                return;

                        getrange(opt,MAXLINES);
```

```
                        break;
                }

                fprintf(stdout,"Incorrect response.\n");

        }
        while (TRUE);

        if (opt->abortop)
                return;

        debgtask = 1;

        fprintf(stdout,"Regnum    Contents\n\n");

        if (opt->rngespan == UNDEFND) {
                tmpreg.val = cnv2addr(_segnum,_offset);
                showmem(&tmpreg,fetchr(&tmpreg,Q));
        }
        else {
                i = 0;

                while((i < opt->rngespan) && (_segnum < _numregseg)) {
                        tmpreg.val = cnv2addr(_segnum,_offset);
                        showmem(&tmpreg,fetchr(&tmpreg,Q));
                        i++;
                        _offset++;
                        if (_offset == _reg[_segnum].num) {
                                _offset = 0;
                                _segnum++;
                        }
                }

        }

        debgtask = 0;
}

/*DISPLSTK()
        function:
                -This function performs the 'display stack' operation.

        interface:
                (p) opt
                (x) _numstkseg/am.h
                (x) _stk[]/am.h
                (g) _segnum
                (x) debgtask/am.h

        called by:
                debug()/debug.c
```

```
    calls:
            cnv2addr()/debugutl.c
            stripblk()/debugutl.c
            get_segm()
            get_ofst()

    errors:

*/

displstk(opt)
OPTION *opt;


{   char *strptr;
    char inpstr[MXDECSTR];
    long stksize = 0;
    int i;
    SAD stktop;

    opt->oprtn = DISPLSTK;

    do {
            fprintf(stdout,"Enter  one of following:\n");
            fprintf(stdout,"\t\t'v'  -  segment value prompt\n");
            fprintf(stdout,"\t\t'@'  -  to abort the operation: \n");
            fscanf(stdin,"%ls",inpstr);

            strptr = stripblk(inpstr);

            if (strptr[0] == '@') {
                    opt->abortop = TRUE;
                    return;
            }

            if (strptr[0] == 'v') {
                    _segnum = get_segm(&opt->abortop,
                                            _numstkseg - 1);
                    break;
            }

            fprintf(stdout,"Incorrect response.\n");

    }
    while (TRUE);

    if (opt->abortop)
            return;

    debgtask = 1;

    _offset = _stk[_segnum].size  - 1;
```

```
    stktop.val = cnv2addr(_segnum,_offset);

    fprintf(stdout,"Top of Stack:\n");
    showmem(&stktop,topstk(&stktop,Q));

    debgtask = 0;
}

/*DISPLBRK()
    function:
        -This function performs the 'display breaks' operation.

    interface:
        (p) opt
        (g) brktable[]
        (g) mt_init
        (g) mt_slots[]
        (g) topslots

    called by:
        debug()/debug.c

    calls:
        stripblk()/debugutl.c
        amdefs()/amcutl.c

    errors:

*/

displbrk(opt)
OPTION *opt;

{   char *strptr;
    char inpstr[MXINPSTR];
    short i;

    opt->oprtn = DISPLBRK;

    do {
        fprintf(stdout,"Enter '!' to continue\n\t\tor\n");
        fprintf(stdout,"'@' to abort the operation: \n");
        fscanf(stdin,"%ls",inpstr);

        strptr = stripblk(inpstr);

        if (strptr[0] == '@') {
            opt->abortop = TRUE;
            return;
        }

        if (strptr[0] == '!')
```

```
                        break;

                fprintf(stdout,"Incorrect response.\n");
        }
        while (TRUE);

        if (!mt_init) {
                for (i = 0;i < MAXBRKS; i++) {
                        mt_slots[i] = i;
                        brktable[i].memaddr  = 0;
                        brktable[i].opcdval = UNDEFND;
                }

                topslot = MAXBRKS - 1;
                mt_init  = TRUE;
        }

        fprintf(stdout,"\t\t************************\n");
        fprintf(stdout,"\t\t*                      *\n");
        fprintf(stdout,"\t\t*     BREAKPOINTS      *\n");
        fprintf(stdout,"\t\t*                      *\n");
        fprintf(stdout,"\t\t************************\n\n");
        fprintf(stdout,"BRKNUM    MEMADDR   OPCODE\n");

        for(i = 0; i < MAXBRKS; i++) {
                if (brktable[i].opcdval == UNDEFND)
                        fprintf(stdout,"%6d\n",i);
                else {
                        fprintf(stdout,"%6d    ", i);
                        fprintf(stdout,"%08lx    ", brktable[i].memaddr);
                        fprintf(stdout,"%s\n",  amdefs(brktable[i].opcdval));
                }
        }

}

/*DISPL_PC()
        function:
                -This function performs the 'display program counter'
                operation.

        interface:
                (p) opt
                (x) _pc./am.h
                (g) _segnum
                (g) _offset

        called by:
                debug()/debug.c

        calls:
                stripblk()/debugutl.c
```

```
    errors:

*/

displ_pc(opt)
OPTION *opt;

{   char *strptr;
    char inpstr[MXINPSTR];

    opt->oprtn = DISPL_PC;

    do {
        fprintf(stdout,"Enter  one of following:\n");
        fprintf(stdout,"\t\t'!'  to confirm display pc \n");
        fprintf(stdout,"\t\t'@'  to abort the operation \n");
        fscanf(stdin,"%ls",inpstr);

        strptr = stripblk(inpstr);

        if (strptr[0] == '@') {
            opt->abortop = TRUE;
            return;
        }

        if (strptr[0] == '!') {
           _segnum = (_pc.val & X_SEGMSK) >> X_SEGSFT;
           _offset = _pc.val & X_ADRMSK;
           fprintf(stdout,"\n\t\tPRGM  COUNTER in segment %ld ",
                                        _segnum);
           fprintf(stdout,"at  offset %ld.\n",_offset);
           return;
        }

        fprintf(stdout,"Incorrect  response.\n");
    }
    while (TRUE);
}

/*GOEXEC()
    function:
        -This function initiates  the rest of the prompts for the
        'go execution' operation.

    interface:
        (p) opt
        (x) left2do/am.h
        (x) debgcntl/am.h

    called by:
        debug()/debug.c
```

```
    calls:
            str2dec()/debugutl.c
            stripblk()/debugutl.c

    errors:

*/

goexec(opt)
OPTION *opt;

{   char inpstr[MXDECSTR];
    char *str;
    BOOLN validnum;
    long number;

    opt->oprtn = GOEXEC;

    do {
            fprintf(stdout,"Enter  one of choices below:\n");
            fprintf(stdout,"\t\tDecimal  range btwn 1 and %d\n",MAXEXECS);
            fprintf(stdout,"\t\t'!'  - uncontrolled go\n");
            fprintf(stdout,"\t\t'@'  - to abort the operation\n\n");
            fscanf(stdin,"%ls",inpstr);

            str = stripblk(inpstr);

            if (str[0] == '@') {
                    opt->abortop = TRUE;
                    return;
            }

            if (str[0] == '!')
                    return;

            str2dec(str,&validnum,&number);
            if (!validnum) {
                    fprintf(stdout,"Invalid  number entered.\n");
                    continue;
            }

            if ((number < 1) || (number > MAXEXECS)) {
                    fprintf(stdout,"Number  out of range.\n");
                    continue;
            }

            left2do = number;
            debgcntl = 1;
            return;

    }
```

```
    while(TRUE);
}

/*HELP()
    function:
        -This function displays the debugger commands available.

    interface:
        (p) opt

    called by:
        debug()/debug.c

    calls:

    errors:

*/

help(opt)
OPTION *opt;

{   char str[MXINPSTR];

    opt->oprtn = HELP;
    fprintf(stdout,"\t\t   Debugger Commands          \n");
    fprintf(stdout,"\t\t**************************\n");
    fprintf(stdout,"'d'(isplay)\t'm'(emory),  {'*' | seg:offset}, span\n");
    fprintf(stdout,"          \t'r'(egister), seg:offset, span\n");
    fprintf(stdout,"          \t's'(tack - top only), seg\n");
    fprintf(stdout,"          \t'b'(reaks - all)\n");
    fprintf(stdout,"          \t'*'(program counter)\n");
    fprintf(stdout,"'g'(o)     \t{'!' | n <instrs>}\n");
    fprintf(stdout,"'?'(list  available debug commands)\n");
    fprintf(stdout,"'s'(et)    \t'm'(emory), seg:offset, val_type, val\n");
    fprintf(stdout,"          \t'r'(egister), seg:offset, val_type, val\n");
    fprintf(stdout,"          \t's'(tacktop, seg, val_type, val\n");
    fprintf(stdout,"          \t'b'(reak), seg:offset\n");
    fprintf(stdout,"          \t'*'(program counter), seg:offset\n");
    fprintf(stdout,"'t'(race)  \t'l'(on)        < TRACE\n");
    fprintf(stdout,"          \t'z'(off)          STARTED\n");
    fprintf(stdout,"          \t for n instrs       BY 'GO'>\n");
    fprintf(stdout,"'q'(uit  debug and halt execution)\n");
    fprintf(stdout,"Legend:  | - or, [] - optional, <> - comment ");
    fprintf(stdout," {} - Must choose an item.\n\n");
    fprintf(stdout,"Enter  non-blank char to continue.\n");
    fscanf(stdin,"%ls",str);
}
```

```
/*REMOVBRK()
      function:
            -This function removes a breakpoint at a specified
            memory location.

      interface:
            (p) opt
            (x) debgtask/am.h
            (g) mt_init
            (g) brktable[]
            (g) topslot
            (g) mt_slots[]
            (x) Q/am.h

      called by:
            debug()/debug.c

      calls:
            stripblk()/debugutl.c
            str2dec()/debugutl.c
            fetchm()/amstate.c
            storem()/amstate.c
            cnv2addr()/debugutl.c

      errors:

*/

removbrk(opt)
OPTION *opt;

{     char *strptr;
      char inpstr[MXDECSTR];
      int i;
      long number;
      BOOLN validnum;
      VAL *v;
      MAD m;
      BREAKS *brkptr;


      opt->oprtn = REMOVBRK;

      do {
            fprintf(stdout,"Enter  decimal break number ");
            fprintf(stdout,"between  0 and %d\n\t\tor\n",MAXBRKS-1);
            fprintf(stdout,"'@'  to abort the operation: \n");
            fscanf(stdin,"%4s",inpstr);

            strptr = stripblk(inpstr);
```

```
        if (strptr[0] == '@') {
                opt->abortop = TRUE;
                return;
        }

        str2dec(strptr,&validnum,&number);
        if (!validnum) {
                fprintf(stdout,"Invalid  number entered.\n");
                continue;
        }

        if ((number < 0) || (number > (MAXBRKS - 1))) {
                fprintf(stdout,"Number  out of range.\n");
                continue;
        }

        opt->rngebegn = number;
        break;
}
while (TRUE);

if (!mt_init) {
        for (i = 0;i < MAXBRKS; i++) {
                mt_slots[i] = i;
                brktable[i].memaddr  = 0;
                brktable[i].opcdval = UNDEFND;
        }
        topslot = MAXBRKS - 1;
        mt_init  = TRUE;
}

if (topslot > MAXBRKS - 2) {
        fprintf(stdout,"Break  Table empty\n");
        return;
}

if (brktable[opt->rngebegn].opcdval == UNDEFND) {
        fprintf(stdout,"Breakpoint  not in Table.\n");
        return;
}

debgtask = 1;

brkptr = &brktable[opt->rngebegn];
m.val = brkptr->memaddr;
v = fetchm(&m,Q);
v->instrval.val[0].opcdval = brkptr->opcdval;
storem(v,&m,Q);
brkptr->memaddr  = 0;
brkptr->opcdval = UNDEFND;
mt_slots[++topslot] = opt->rngebegn;
```

```
    debgtask = 0;
}
/*TRACEOP()
    function:
        -This function initiates  the rest of the prompts for the
        the 'traceop' operation.

    interface:
        (p) opt
        (x) debgcntl/am.h
        (x) left2do/am.h
        (x) dbgtrace/am.h

    called by:
        debug()/debug.c

    calls:
        stripblk()/debugopr.c
        str2dec()/debugopr.c

    errors:

*/

traceop(opt)
OPTION *opt;

{   char *strptr;
    char inpstr[MXDECSTR];
    int i;
    long number;
    BOOLN validnum;

    opt->oprtn = TRACEOP;

    do {
        fprintf(stdout,"Enter  one of following:\n");
        fprintf(stdout,"\t\tDecimal  number between 1 and ");
        fprintf(stdout,"%d\n",MAXLINES);
        fprintf(stdout,"\t\t'!'  for 'trace on'\n");
        fprintf(stdout,"\t\t'z'  for 'trace off'\n");
        fprintf(stdout,"\t\t'@'  to abort the operation:\n");
        fscanf(stdin,"%10s",inpstr);

        strptr = stripblk(inpstr);

        if (strptr[0] == '@') {
            opt->abortop = TRUE;
            return;
        }

        if (strptr[0] == '!') {
```

```
                opt->rngespan = TRACEON;
                dbgtrace = 1;
                debgcntl = 0;
                return;
        }

        if (strptr[0] == 'z') {
                opt->rngespan = TRACEOFF;
                dbgtrace = 0;
                debgcntl = 0;
                return;
        }

        str2dec(strptr,&validnum,&number);

        if (!validnum) {
                fprintf(stdout,"Invalid  number entered.\n");
                continue;
        }

        if ((number < 1) || (number > MAXLINES)) {
                fprintf(stdout,"Number  out of range.\n");
                continue;
        }

        left2do = number;
        dbgtrace = 1;
        debgcntl = 1;
        return;
    }
    while (TRUE);

}
/*QUITDEBG()
    function:
        -Uponuser confirmation, this function exits the debug-
        ger and halts program execution.

    interface:
        (p) opt

    called by:
        debug()/debug.c

    calls:
        stripblk()/debugopr.c
        exit()

    errors:

*/
```

```
quitdebg(opt)
OPTION *opt;

{     char *strptr;
      char inpstr[MXINPSTR];

      opt->oprtn = QUITDEBG;

      do {
            fprintf(stdout,"Enter  one of following:\n");
            fprintf(stdout,"\t\t'!'  to END DEBUG AND EXECUTION\n");
            fprintf(stdout,"\t\t'@'  to abort operation:\n");
            fscanf(stdin,"%ls",inpstr);

            strptr = stripblk(inpstr);

            if (strptr[0] == '@') {
                  opt->abortop = TRUE;
                  return;
            }

            if (strptr[0] == '!') {
              fprintf(stdout,"Exiting  Debugger, Halting Execution.\n");
              exit (0);
            }

            fprintf(stdout,"Incorrect  response.\n");

      }
      while (TRUE);

}
/*GET_BOOL()
      function:
            -This function initializes  the boolval structure, prompt-
            ting the user to enter the actual value part of the
            structure.

      interface:
            (p) v
            (p) abortop

      called by:
            getvalue()

      calls:
            stripblk()/debugutl.c

      errors:

*/
```

```
get_bool(v,abortop)
VAL *v;
BOOLN *abortop;

{     char *strptr;
      char inpstr[MXINPSTR];

      v->boolval.type = V_BOOL;

      do {
            fprintf(stdout,"Enter  letter  of your choice:\n");
            fprintf(stdout,"\t\tt(rue)\n\t\tf(alse)\n");
            fprintf(stdout,"\t\t@(abort  operation\n");
            fscanf(stdin,"%ls",inpstr);

            strptr  = stripblk(inpstr);

            switch(strptr[0]) {
                  case 't':  v->boolval.val = 't';
                             return;
                  case 'f':  v->boolval.val = 'f';
                             return;
                  case '@':  *abortop = TRUE;
                             return;
                  default:   fprintf(stdout,"Bad  response.\n");

            }
      }
      while (TRUE);

}

/*GET_NAT()
      function:
            -This function initializes  the natval structure, prompt-
            ting the user to enter the actual value part of the
            structure.

      interface:
            (p) v
            (p) abortop

      called by:
            getvalue()

      calls:
            stripblk()/debugutl.c
            str2dec()/debugutl.c

      errors:

*/
```

```
get_nat(v,abortop)
VAL *v;
BOOLN *abortop;

{     long number;
      BOOLN validnum;
      char *strptr;
      char inpstr[MXDECSTR];

      v->natval.type = V_NAT;

      do {
            fprintf(stdout,"Enter  decimal number between\n");
            fprintf(stdout,"\t\t0  and 65535\nor\n");
            fprintf(stdout,"\t\t'@'  to abort the operation:\n");
            fscanf(stdin,"%10s",inpstr);

            strptr = stripblk(inpstr);

            if (strptr[0] == '@') {
                  *abortop = TRUE;
                  return;
            }

            str2dec(strptr,&validnum,&number);

            if (!validnum) {
                  fprintf(stdout,"Invalid  number entered.\n");
                  continue;
            }

            if ((number < 0) || (number > 65535)) {
                  fprintf(stdout,"Number  out of range.\n");
                  continue;
            }

            v->natval.val = number;
            return;
      }
      while (TRUE);

}
/*GET_INT()
      function:
            -This function initializes  the intval structure, promp-
            ting the user to enter the actual value part of the
            structure.

      interface:
            (p) v
            (p) abortop
```

```
        called by:
                getvalue()

        calls:
                stripblk()/debugutl.c
                str2dec()/debugutl.c
        errors:

*/

get_int(v,abortop)
VAL *v;
BOOLN *abortop;

{       long number;
        BOOLN validnum;
        char *strptr;
        char inpstr[MXDECSTR];

        v->intval.type  = V_INT;

        do {
                fprintf(stdout,"Enter  decimal number between\n\n");
                fprintf(stdout,"\t\t  -2147483647  & 2147483647  (no ','):\n");
                fprintf(stdout,"or\n\t\t'@'  to abort the operation:\n");
                fscanf(stdin,"%ls",inpstr);

                strptr = stripblk(inpstr);

                if (strptr[0] == '@') {
                        *abortop = TRUE;
                        return;
                }

                str2dec(strptr,&validnum,&number);

                if (!validnum) {
                        fprintf(stdout,"Invalid  number entered.\n");
                        continue;
                }

                v->intval.val  = number;
                return;
        }
        while (TRUE);

}
/*GET_CHAR()
        function:
                -This function initializes  the charval structure, promp-
                ting the user to enter the actual value part of the
```

```
            structure.

      interface:
            (p) v
            (p) abortop

      called by:
            getvalue()

      calls:
            getchar()/ *system*
      errors:

*/

get_char(v,abortop)
VAL *v;
BOOLN *abortop;

{     char ch;

      v->charval.type = V_CHAR;

      fprintf(stdout,"Enter  character\n\t\tor\n");
      fprintf(stdout,"'@'  to abort the operation:\n");

      ch = getchar(); /* MUST be done to CLEAR LAST NEWLINE CHARACTER
                         user typed. */

      ch = getchar();

      if (ch == '@') {
            *abortop =TRUE;
            return;
      }

      v->charval.val = ch;
      return;

}

/*GET_CSTR()
      function:
            -This function initializes  the cstrval structure, promp-
            ting the user to enter the actual value part of the
            structure.

      interface:
            (p) v
            (p) abortop

      called by:
```

```
            getvalue()

      calls:
            pmalloc()/amstate.c
            getchar()/ *system*
            gets()/ *system*
            strcpy()/ *system*

      errors:

*/

get_cstr(v,abortop)
VAL *v;
BOOLN *abortop;

{     char str[81];
      char *ptr;
      char *gets();
      char ch;

      v->cstrval.type = V_CSTR;

      do {
            fprintf(stdout,"Enter  char string (max 80 chars)\n");
            fprintf(stdout,"\t\tor\n'@'  to abort the operation:\n");

            ch = getchar(); /* MUST be done to CLEAR LAST NEWLINE
                              character user typed. */

            ptr = gets(str);

            if (ptr != NULL) {
                  if (str[0] == '@') {
                        *abortop = TRUE;
                        return;
                  }

                  v->cstrval.val = pmalloc(strlen(str) +1);
                  strcpy(v->cstrval.val, str);
                  return;
            }

            fprintf(stdout,"Error  in reading string.\n");
      }
      while (TRUE);

}
/*GET_MAD()
      function:
            -This function initializes  the madval structure, promp-
            ting the user to enter the actual value part of the
```

```
            structure.

    interface:
            (p) v
            (p) abortop

    called by:
            getvalue()

    calls:
            cnv2addr()/debugutl.c
            stripblk()/debugutl.c
            get_segm()
            get_ofst()

    errors:

*/

get_mad(v,abortop)
VAL *v;
BOOLN*abortop;

{   char *strptr;
    char inpstr[MXINPSTR];
    long    _segnum;
    long    _offset;

    v->madval.type = V_MAD;

    do {
            fprintf(stdout,"Enter  one of following:\n");
            fprintf(stdout,"\t\t'v'  - memaddr value prompt\n");
            fprintf(stdout,"\t\t'@'  - to abort the operation: \n");
            fscanf(stdin,"%ls",inpstr);

            strptr = stripblk(inpstr);

            if (strptr[0] == '@') {
                    *abortop = TRUE;
                    return;
            }

            if (strptr[0] == 'v') {
                    _segnum = get_segm(abortop,_numusrseg - 1);

                    if (*abortop)
                            return;

                    _offset = get_ofst(abortop,
                                            _mem[_segnum].size - 1);
```

```
                        if (*abortop)
                                return;

                        v->madval.val = cnv2addr(_segnum, _offset);

                        return;
                }

                fprintf(stdout,"Incorrect  response.\n");

        }
        while (TRUE);
}

/*GET_RAD()
        function:
                -This function initializes  the radval structure, promp-
                ting the user to enter the actual value part of the
                structure.

        interface:
                (p) v
                (p) abortop

        called by:
                getvalue()

        calls:
                cnv2addr()/debugutl.c
                stripblk()/debugutl.c
                get_segm()
                get_ofst()

        errors:

*/

get_rad(v,abortop)
VAL *v;
BOOLN *abortop;

{       char *strptr;
        char inpstr[MXINPSTR];
        long    _segnum;
        long    _offset;

        v->radval.type = V_RAD;

        do {
                fprintf(stdout,"Enter  one of following:\n");
                fprintf(stdout,"\t\t'v'  -  regaddr value prompt\n");
                fprintf(stdout,"\t\t'@'  -  to abort the operation: \n");
```

```
            fscanf(stdin,"%ls",inpstr);

            strptr = stripblk(inpstr);

            if (strptr[0] == '@') {
                    *abortop = TRUE;
                    return;
            }

            if (strptr[0] == 'v') {
                    _segnum = get_segm(abortop,_numregseg - 1);

                    if (*abortop)
                            return;

                    _offset = get_ofst(abortop,_reg[_segnum].num - 1);

                    if (*abortop)
                            return;

                    v->radval.val = cnv2addr(_segnum, _offset);

                    return;
            }

            fprintf(stdout,"Incorrect response.\n");

    }
    while (TRUE);
}

/*GET_SAD()
    function:
            -This function initializes the sadval structure, prompt-
            ting the user to enter the actual value part of the
            structure.

    interface:
            (p) v
            (p) abortop

    called by:
            getvalue()

    calls:
            cnv2addr()/debugutl.c
            stripblk()/debugutl.c
            get_segm()
            get_ofst()

    errors:
```

```
*/

get_sad(v,abortop)
VAL *v;
BOOLN *abortop;

{     char *strptr;
      char inpstr[MXINPSTR];
      long    _segnum;
      long    _offset;

      v->sadval.type = V_SAD;

      do {
            fprintf(stdout,"Enter  one of following:\n");
            fprintf(stdout,"\t\t'v'  -  stkaddr value prompt\n");
            fprintf(stdout,"\t\t'@'  -  to abort the operation: \n");
            fscanf(stdin,"%ls",inpstr);

            strptr = stripblk(inpstr);

            if (strptr[0] == '@') {
                  *abortop = TRUE;
                  return;
            }

            if (strptr[0] == 'v') {
                  _segnum = get_segm(abortop, _numstkseg - 1);

                  if (*abortop)
                        return;

                  _offset = get_ofst(abortop,_stk[_segnum].size  - 1);

                  if (*abortop)
                        return;

                  v->sadval.val = cnv2addr(_segnum, _offset);

                  return;
            }

            fprintf(stdout,"Incorrect  response.\n");

      }
      while (TRUE);
}

/*GET_FILE()
      function:
            -This function initializes  the fileval structure, promp-
            ting the user to enter the actual value part of the
```

```
            structure.

    interface:
            (p) v
            (p) abortop

    called by:
            getvalue()

    calls:
            stripblk()/debugutl.c
            str2dec()/debugutl.c

    errors:

*/

get_file(v,abortop)
VAL *v;
BOOLN *abortop;

{   long number;
    BOOLN validnum;
    char *strptr;
    char inpstr[MXDECSTR];

    v->fileval.type = V_FILE;

    do {
            fprintf(stdout,"Enter  decimal number ");
            fprintf(stdout,"between  0 and %d\n\t\tor\n",_numfiles - 1);
            fprintf(stdout,"'@'  to abort the operation:\n");
            fscanf(stdin,"%11s",inpstr);

            strptr = stripblk(inpstr);

            if (strptr[0] == '@') {
                    *abortop = TRUE;
                    return;
            }

            str2dec(strptr,&validnum,&number);

            if (!validnum) {
                    fprintf(stdout,"Invalid  number entered.\n");
                    continue;
            }

            if ((number < 0) || (number > _numfiles - 1)) {
                    fprintf(stdout,"Number  out of range.\n");
                    continue;
            }
```

```
            v->fileval.val = number;
            return;
        }
        while (TRUE);

}

/*GET_INST()
        function:
            -This function initializes the instrval structure, prompting the user to enter the actual value part of the
            structure. It resets global file var 'inst_get' to FALSE
            just before function termination.

        interface:
            (p) v
            (p) abortop
            (g) inst_get

        called by:
            getvalue()

        calls:
            getopnd()/aminstr.c
            getopcode()/aminstr.c
            getvalue()   /* Note that this is a recursive call */
            pmalloc()/amstate.c
            str2hex()/debugutl.c
            stripblk()/debugutl.c

        errors:

*/

get_inst(v,abortop)
VAL *v;
BOOLN *abortop;

{       long number;
        BOOLN validnum;
        char *strptr;
        char inpstr[MXHEXSTR];
        int i;
        int j;
        VAL *p;

        v->instrval.type = V_INSTR;
        inst_get = TRUE;

        do {
            fprintf(stdout,"Enter  HEX opcode\n\t\tor\n");
```

```
            fprintf(stdout,"'@'  to abort the operation:\n");
            fscanf(stdin,"%4s",inpstr);

            strptr = stripblk(inpstr);

            if (strptr[0] == '@') {
                    *abortop = TRUE;
                    inst_get = FALSE;
                    return;
            }

            str2hex(strptr,&validnum,&number);

            if (!validnum) {
                    fprintf(stdout,"Invalid  number entered.\n");
                    continue;
            }

            if ((number < 0) || (number > 0x6fff)) {
                    fprintf(stdout,"Invalid  opcode.\n");
                    continue;
            }

            if (getopcode(number) == IL_DBG) {
                    fprintf(stdout,"Breakpoint  opcodeentered.\n");
                    fprintf(stdout,"Opcode can't be entered.\n\n");
                    continue;
            }

            j = getopnd(number);
            v->instrval.val = (VAL *) pmalloc(sizeof(VAL)* j);
            v->instrval.val[0].opcdval = number;
            p = v->instrval.val;

            for (i = 1; i < j; i++) {
                    fprintf(stdout,"\t\t  ***    Entering Operand #%d",i);
                    fprintf(stdout,"     ***\n");
                    getvalue(&p[i], abortop);

                    if (*abortop) {
                            inst_get = FALSE;
                            return;
                    }
            }

            inst_get = FALSE;
            return;

    }
    while (TRUE);

}
```

```
/*GET_MOP()
     function:
          -This function initializes  the mopval structure, promp-
          ting the user to enter the actual value part of the
          structure.

     interface:
          (p) v
          (p) abortop

     called by:
          getvalue()

     calls:
          str2hex()/debugutl.c
          stripblk()/debugutl.c

     errors:

*/

get_mop(v,abortop)
VAL *v;
BOOLN *abortop;

{    long number;
     BOOLN validnum;
     char *strptr;
     char inpstr[MXHEXSTR];

     v->mopval.type = V_MOP;

     do {
          fprintf(stdout,"Enter  HEX number ");
          fprintf(stdout,"between  0 and ffff\n\t\tor\n");
          fprintf(stdout,"'@'  to abort the operation:\n");
          fscanf(stdin,"%4s",inpstr);

          strptr  = stripblk(inpstr);

          if (strptr[0] == '@') {
               *abortop = TRUE;
               return;
          }

          str2hex(strptr,&validnum,&number);

          if (!validnum) {
               fprintf(stdout,"Invalid  number entered.\n");
               continue;
          }
```

```
            if ((number < 0) || (number > 65535)) {
                    fprintf(stdout,"Number  out of range.\n");
                    continue;
            }

            v->mopval.val = number;
            return;
    }
    while (TRUE);

}

/*GET_DOP()
    function:
            -This function initializes  the dopval structure, promp-
            ting the user to enter the actual value part of the
            structure.

    interface:
            (p) v
            (p) abortop

    called by:
            getvalue()

    calls:
            str2hex()/debugutl.c
            stripblk()/debugutl.c

    errors:

*/

get_dop(v,abortop)
VAL *v;
BOOLN *abortop;

{   long number;
    BOOLN validnum;
    char *strptr;
    char inpstr[MXHEXSTR];

    v->dopval.type = V_DOP;

    do {
            fprintf(stdout,"Enter  HEX number ");
            fprintf(stdout,"between  0 and ffff\n\t\tor\n");
            fprintf(stdout,"'@'  to abort the operation:\n");
            fscanf(stdin,"%4s",inpstr);

            strptr = stripblk(inpstr);
```

```
            if (strptr[0] == '@') {
                    *abortop = TRUE;
                    return;
            }

            str2hex(strptr,&validnum,&number);

            if (!validnum) {
                    fprintf(stdout,"Invalid  number entered.\n");
                    continue;
            }

            if ((number < 0) || (number > 65535)) {
                    fprintf(stdout,"Number  out of range.\n");
                    continue;
            }

            v->dopval.val = number;
            return;
    }
    while (TRUE);

}

/*GET_ROP()
    function:
            -This function initializes  the ropval structure, promp-
            ting the user to enter the actual value part of the
            structure.

    interface:
            (p) v
            (p) abortop

    called by:
            getvalue()

    calls:
            str2hex()/debugutl.c
            stripblk()/debugutl.c

    errors:

*/

get_rop(v,abortop)
VAL *v;
BOOLN *abortop;

{   long number;
    BOOLN validnum;
```

```
    char *strptr;
    char inpstr[MXHEXSTR];

    v->ropval.type = V_ROP;

    do {
            fprintf(stdout,"Enter  HEX number ");
            fprintf(stdout,"between  0 and ffff\n\t\tor\n");
            fprintf(stdout,"'@'  to abort the operation:\n");
            fscanf(stdin,"%4s",inpstr);

            strptr = stripblk(inpstr);

            if (strptr[0] == '@') {
                    *abortop = TRUE;
                    return;
            }

            str2hex(strptr,&validnum,&number);

            if (!validnum) {
                    fprintf(stdout,"Invalid  number entered.\n");
                    continue;
            }

            if ((number < 0) || (number > 65535)) {
                    fprintf(stdout,"Number  out of range.\n");
                    continue;
            }

            v->ropval.val = number;
            return;
    }
    while (TRUE);

}

/*GET_BOP()
    function:
            -This function initializes  the bopval structure, promp-
            ting the user to enter the actual value part of the
            structure.

    interface:
            (p) v
            (p) abortop

    called by:
            getvalue()

    calls:
            str2hex()/debugutl.c
```

```
            stripblk()/debugutl.c

    errors:

*/

get_bop(v,abortop)
VAL *v;
BOOLN *abortop;

{   long number;
    BOOLN validnum;
    char *strptr;
    char inpstr[MXHEXSTR];

    v->bopval.type = V_BOP;

    do {
            fprintf(stdout,"Enter  HEX number ");
            fprintf(stdout,"between  0 and ffff\n\t\tor\n");
            fprintf(stdout,"'@'  to abort the operation:\n");
            fscanf(stdin,"%4s",inpstr);

            strptr  = stripblk(inpstr);

            if (strptr[0] == '@') {
                    *abortop = TRUE;
                    return;
            }

            str2hex(strptr,&validnum,&number);

            if (!validnum) {
                    fprintf(stdout,"Invalid  number entered.\n");
                    continue;
            }

            if ((number < 0) || (number > 65535)) {
                    fprintf(stdout,"Number  out of range.\n");
                    continue;
            }

            v->bopval.val = number;
            return;
    }
    while (TRUE);

}

/*GETVALUE()
    function:
            -This function prompts the user for value of type VAL
```

```
            which can be placed into memory, a register or the
            stack.

    interface:
            (p) v
            (p) abortop
            (g) inst_get

    called by:
            setmemr()
            setregr()
            setstk()
            get_inst() /* Note -- indirect recursive call!! */

    calls:
            str2dec()/debugutl.c
            stripblk()/debugutl.c
            get_bool()
            get_nat()
            get_int()
            get_char()
            get_cstr()
            get_mad()
            get_mad()
            get_rad()
            get_sad()
            get_file()
            get_inst()
            get_mop()
            get_dop()
            get_rop()
            get_bop()

    errors:

*/

getvalue(v,abortop)
VAL *v;
BOOLN *abortop;

{   char *strptr;
    char inpstr[MXDECSTR]; /* MXDECSTR = 12 */
    int number;
    BOOLN validnum;

    do {

            fprintf(stdout,"\t\t  **** Entering Value to be Stored
****\n\n");
            fprintf(stdout,"Enter  number ");
            fprintf(stdout,"besides  type desired:\n\n");
```

```
fprintf(stdout,"\t  1 - BOOL \t 2 - NAT  \t 3 - INT  ");
fprintf(stdout,"\t  4 - CHAR\n\t 5 - CSTR \t 6 - MAD  ");
fprintf(stdout,"\t  7 - RAD  \t 8 - SAD\n\t 9 - FILE ");

if (!inst_get) {
 fprintf(stdout,"\t10   - INSTR\t11   - MOP  \t12  - DOP\n");
 fprintf(stdout,"\t13   - ROP  \t14  - BOP  \t @ - abort op\n");
}
else {
 fprintf(stdout,"\t11   - MOP  \t12  - DOP  \t13  - ROP\n");
 fprintf(stdout,"\t14   - BOP  \t @ - abort op\n");
}

fscanf(stdin,"%11s",inpstr);

strptr = stripblk(inpstr);

if (strptr[0] == '@') {
        *abortop = TRUE;
        return;
}

str2dec(strptr,&validnum,&number);
if (!validnum) {
        fprintf(stdout,"Invalid  number entered.\n");
        continue;
}

switch(number) {
        case 1:   get_bool(v,abortop);
                  return;
        case 2:   get_nat(v,abortop);
                  return;
        case 3:   get_int(v,abortop);
                  return;
        case 4:   get_char(v,abortop);
                  return;
        case 5:   get_cstr(v,abortop);
                  return;
        case 6:   get_mad(v,abortop);
                  return;
        case 7:   get_rad(v,abortop);
                  return;
        case 8:   get_sad(v,abortop);
                  return;
        case 9:   get_file(v,abortop);
                  return;

        case 10:  if (!inst_get) {
                        get_inst(v,abortop);
                        return;
                  }
```

```
                                        fprintf(stdout,"Incorrect number.\n");
                                        break;

                        case 11:    get_mop(v,abortop);
                                        return;
                        case 12:    get_dop(v,abortop);
                                        return;
                        case 13:    get_rop(v,abortop);
                                        return;
                        case 14:    get_bop(v,abortop);
                                        return;

                        default:    fprintf(stdout,"Incorrect number.\n");

                }
        }
        while (TRUE);

}

/*SETMEMR()
        function:
                -This function performs the 'set memory' operation.

        interface:
                (p) opt
                (g) _segnum
                (g) -offset
                (x) debgtask/am.h

        called by:
                debug()/debug.c

        calls:
                stripblk()/debugutl.c
                getvalue()
                fetchm()/amstate.c
                storem()/amstate.c
                pmalloc()/amstate.c
                fmalloc()/amstate.c
                cnv2addr()/debugutl.c
                getopcode()/aminstr.c

        errors:

*/

setmemr(opt)
OPTION *opt;

{       char *strptr;
        char inpstr[MXINPSTR];
```

```
BOOLN op_abort = FALSE;
VAL *v;
MAD m;
short brknum;

opt->oprtn = SETMEMR;

do {
        fprintf(stdout,"Enter  one of following:\n");
        fprintf(stdout,"\t\t'v'  - memaddr value prompt\n");
        fprintf(stdout,"\t\t'@'  - to abort the operation: \n");
        fscanf(stdin,"%ls",inpstr);

        strptr = stripblk(inpstr);

        if (strptr[0] == '@') {
                opt->abortop = TRUE;
                return;
        }

        if (strptr[0] == 'v') {
                _segnum = get_segm(&opt->abortop,
                                                _numusrseg - 1);

                if (opt->abortop)
                        return;

                _offset = get_ofst(&opt->abortop,
                                _mem[_segnum].size - 1);

                if (opt->abortop)
                        return;

                opt->val = (VAL *) pmalloc(sizeof(VAL));
                getvalue(opt->val,&op_abort);

                break;
        }

        fprintf(stdout,"Incorrect  response.\n");

}
while (TRUE);

if (opt->abortop = op_abort) /* Assignment intended!! */
        return;

debgtask = 1;

m.val = cnv2addr(_segnum, _offset);
v = fetchm(&m,Q);
```

```
    if (v->type == V_INSTR) {
            if (getopcode(v->instrval.val[0].opcdval) == IL_DBG) {
                    do {
                     fprintf(stdout,"Breakpoint  at memaddr.\n");
                     fprintf(stdout,"Enter  '!' to confirm set\n");
                     fprintf(stdout,"\t\tor\n'@'  to abort op:\n");
                     fscanf(stdin,"%1s",inpstr);

                     strptr = stripblk(inpstr);

                     if (strptr[0] == '@') {
                             opt->abortop = TRUE;
                             debgtask = 0;
                             return;
                     }

                     if (strptr[0] == '!') {
                             brknum=getopnd(v->instrval.val[0].opcdval);
                             storem(opt->val,&m,Q);
                             fmalloc(opt->val);
                             brktable[brknum].memaddr  = 0;
                             brktable[brknum].opcdval  = UNDEFND;
                             mt_slots[++topslot]  = brknum;
                             debgtask = 0;
                             return;
                     }

                     fprintf(stdout,"Incorrect  response.\n");

                    }
                    while (TRUE);
            }
    }


    storem(opt->val,&m,Q);
    fmalloc(opt->val);
    debgtask = 0;
}

/*SETREGR()
    function:
            -This function performs the 'set register' operation.

    interface:
            (p) opt
            (g) _segnum
            (g) -offset

    called by:
            debug()/debug.c
```

```
    calls:
            stripblk()/debugutl.c
            getvalue()
            pmalloc()/amstate.c
            fmalloc()/amstate.c
            cnv2addr()/debugutl.c

    errors:

*/

setregr(opt)
OPTION *opt;

{   char *strptr;
    char inpstr[MXINPSTR];
    BOOLN op_abort = FALSE;
    RADr;


    opt->oprtn = SETREGR;

    do {
            fprintf(stdout,"Enter  one of following:\n");
            fprintf(stdout,"\t\t'v'  - regaddr value prompt\n");
            fprintf(stdout,"\t\t'@'  - to abort the operation: \n");
            fscanf(stdin,"%ls",inpstr);

            strptr = stripblk(inpstr);

            if (strptr[0] == '@') {
                    opt->abortop = TRUE;
                    return;
            }

            if (strptr[0] == 'v') {
                    _segnum = get_segm(&opt->abortop,
                                            _numregseg - 1);

                    if (opt->abortop)
                            return;

                    _offset = get_ofst(&opt->abortop,
                                            _reg[_segnum].num - 1);

                    if (opt->abortop)
                            return;

                    opt->val = (VAL *) pmalloc(sizeof(VAL));
                    getvalue(opt->val,&op_abort);

                    break;
```

```
            }

            fprintf(stdout,"Incorrect response.\n");

        }
        while (TRUE);

        if (opt->abortop = op_abort) /* Assignment intended!! */
            return;

        r.val = cnv2addr(_segnum, _offset);
        storer(opt->val,&r,Q);
        fmalloc(opt->val);

}

/*SETSTK()
        function:
            -This function performs the 'set stack' operation.

        interface:
            (p) opt
            (g) _segnum

        called by:
            debug()/debug.c

        calls:
            stripblk()/debugutl.c
            getvalue()
            pmalloc()/amstate.c
            fmalloc()/amstate.c
            cnv2addr()/debugutl.c

        errors:

*/

setstk(opt)
OPTION *opt;

{       char *strptr;
        char inpstr[MXINPSTR];
        BOOLN op_abort = FALSE;
        SAD s;

        opt->oprtn = SETSTK;

        do {
            fprintf(stdout,"Enter  one of following:\n");
            fprintf(stdout,"\t\t'v'  -  regaddr value prompt\n");
            fprintf(stdout,"\t\t'@'  -  to abort the operation: \n");
```

```
            fscanf(stdin,"%ls",inpstr);

            strptr = stripblk(inpstr);

            if (strptr[0] == '@') {
                    opt->abortop = TRUE;
                    return;
            }

            if (strptr[0] == 'v') {
                    _segnum = get_segm(&opt->abortop,
                                            _numstkseg - 1);

                    if (opt->abortop)
                            return;

                    opt->val = (VAL *) pmalloc(sizeof(VAL));
                    getvalue(opt->val,&op_abort);

                    break;
            }

            fprintf(stdout,"Incorrect response.\n");

    }
    while (TRUE);

    if (opt->abortop = op_abort) /* Assignment intended!! */
            return;

    _offset = _stk[_segnum].size -1;
    s.val = cnv2addr(_segnum, _offset);
    storestk(opt->val,&s);
    fmalloc(opt->val);

}

/*SETBRK()
    function:
            -This function sets a breakpoint at an memaddr with an
            instruction.

    interface:
            (p) opt
            (x) debgtask/am.h
            (g) mt_init
            (g) brktable[]
            (g) topslot
            (g) mt_slots[]
            (g) _segnum
            (g) _offset
```

```
        called by:
                debug()/debug.c

        calls:
                stripblk()/debugutl.c
                getopcode()/aminstr.c
                fetchm()/amstate.c
                storem()/amstate.c
                cnv2addr()/debugutl.c

        errors:

*/

setbrk(opt)
OPTION *opt;

{       char *strptr;
        char inpstr[MXINPSTR];
        short opcode;
        VAL     *v;
        MAD     m;
        short   i;
        BREAKS brk;

        opt->oprtn = SETBRK;

        do {
                fprintf(stdout,"Enter  one of following:\n");
                fprintf(stdout,"\t\t'v'  -  memaddr value prompt\n");
                fprintf(stdout,"\t\t'@'  -  to abort the operation: \n");
                fscanf(stdin,"%ls",inpstr);

                strptr  = stripblk(inpstr);

                if (strptr[0] == '@') {
                        opt->abortop = TRUE;
                        return;
                }

                if (strptr[0] == 'v') {
                        _segnum = get_segm(&opt->abortop,
                                                _numusrseg - 1);

                        if (opt->abortop)
                                return;

                        _offset = get_ofst(&opt->abortop,
                                                _mem[_segnum].size - 1);

                        if (opt->abortop)
                                return;
```

```
                break;
        }

        fprintf(stdout,"Incorrect response.\n");

}
while (TRUE);

if (!mt_init) {
        for (i = 0;i < MAXBRKS; i++) {
                mt_slots[i] = i;
                brktable[i].memaddr  = 0;
                brktable[i].opcdval = UNDEFND;
        }

        topslot = MAXBRKS - 1;
        mt_init  = TRUE;
}

if (topslot < 0) {
        fprintf(stdout,"Break  Table Full.\n");
        return;
}

debgtask = 1;

m.val = cnv2addr(_segnum, _offset);
v = fetchm(&m,Q);

if (v->type != V_INSTR) {
        fprintf(stdout,"Sorry,  non-instr at memaddr.\n");
        debgtask = 0;
        return;
}

if (getopcode(v->instrval.val[0].opcdval) == II_DBG) {
        fprintf(stdout,"Sorry,  Breakpoint already at memaddr.\n");
        debgtask = 0;
        return;
}

brk.opcdval = v->instrval.val[0].opcdval;
brk.memaddr  = m.val;

opcode= ((mt_slots[topslot] << X_OPNDSF) | (II_DBG));
v->instrval.val[0].opcdval = opcode;
brktable[mt_slots[topslot]].memaddr  = brk.memaddr;
brktable[mt_slots[topslot]].opcdval  = brk.opcdval;
topslot--;

storem(v,&m,Q);
```

```
        debgtask = 0;

}
/*SET_PC()
        function:
                -This function performs the 'set program counter'
                operation.

        interface:
                (p) opt
                (x) _numusrseg/am.h
                (x) _pc/am.h
                (x) _mem[]/am.h
                (g) _segnum
                (g) _offset
                (x) debgtask/am.h

        called by:
                debug()/debug.c

        calls:
                stripblk()/debugutl.c
                fetchm()/amstate.c
                cnv2addr()/debugutl.c
                get_segm()
                get_ofst()

        errors:

*/

set_pc(opt)
OPTION *opt;

{       char *strptr;
        char inpstr[MXINPSTR];
        VAL *v;
        MAD m;

        opt->oprtn = SET_PC;

        do {
                fprintf(stdout,"Enter  one of following:\n");
                fprintf(stdout,"\t\t'v'  -  prog cntr value prompt\n");
                fprintf(stdout,"\t\t'@'  -  to abort the operation: \n");
                fscanf(stdin,"%ls",inpstr);

                strptr  = stripblk(inpstr);

                if (strptr[0] == '@') {
                        opt->abortop = TRUE;
```

```
                return;
        }

        if (strptr[0] == 'v') {
                _segnum = get_segm(&opt->abortop,
                                        _numusrseg - 1);

                if (opt->abortop)
                        return;

                _offset = get_ofst(&opt->abortop,
                                        _mem[_segnum].size - 1);

                if (opt->abortop)
                        return;

                break;
        }

        fprintf(stdout,"Incorrect response.\n");

    }
    while (TRUE);

    debgtask = 1;

    m.val = cnv2addr(_segnum, _offset);
    v = fetchm(&m,Q);

    if (v->type != V_INSTR) {
            fprintf(stdout,"Sorry, non-instr at memaddr.\n");
            fprintf(stdout,"Program counter unchanged. \n");
            opt->abortop = TRUE;
            return;
    }

    _pc.val = m.val;

    debgtask = 0;
}
```

```
/*DEBUGUTL.C        : This file contains the utility programs for the AM
                    debugger.

    -AM version 1.0 - Z100

Changes:

*/

#include "amdef.h"
#include "amtype.h"
#include "amextern.h"
#include "debug.h"

/*CNV2ADDR()
    function:
            -This functions converts its parameters into a regular
            segmented memory address.

    interface:
            (p) _segnum
            (p) _offset

    called by:
            displmem()/debugopr.c
            removbrk()/debugopr.c
            setmemr()/debugopr.c
            setbrk()/debugopr.c
            set_pc()/debugopr.c
            get_mad()/debugopr.c
            displreg()/debugopr.c
            setregr()/debugopr.c
            get_rad()/debugopr.c
            displstk()/debugopr.c
            setstk()/debugopr.c
            get_sad()/debugopr.c

    calls:

    errors:

*/

address
cnv2addr(_segnum, _offset)
long _segnum;
long _offset;
```

```
{
    return((_segnum << X_SEGSFT) | _offset);
}
/*STRIPBLK()
    function:
        -This functions strips leading blank characters from a
        character string.

    interface:
        (p) str

    called by:
        displmem()/debugopr.c
        displreg()/debugopr.c
        displbrk()/debugopr.c
        displstk()/debugopr.c
        displ_pc()/debugopr.c
        getopr()/debug.c
        getresrc()/debug.c
        getvalue()/debugopr.c
        goexec()/debugopr.c
        removbrk()/debugopr.c
        setmemr()/debugopr.c
        setregr()/debugopr.c
        setstk()/debugopr.c
        setbrk()/debugopr.c
        set_pc()/debugopr.c
        traceop()/debugopr.c
        quitdebg()/debugopr.c
        get_mad()/debugopr.c
        get_rad()/debugopr.c
        get_sad()/debugopr.c
        get_int()/debugopr.c
        get_bool()/debugopr.c
        get_file()/debugopr.c
        get_mop()/debugopr.c
        get_bop()/debugopr.c
        get_rop()/debugopr.c
        get_dop()/debugopr.c
        get_inst()/debugopr.c
        get_segm()/debugopr.c
        get_ofst()/debugopr.c
        quitdebg()/debugopr.c


    calls:
        strlen()/ *system*
    errors:

*/

char
```

```
*stripblk(str)
char *str;
{
     int index = 0;
     int lngth;

     lngth = strlen(str);

     while ((index < lngth) && (str[index] == ' '))
          index++;

     return(str + index);
}
/*STR2HEX()
     function:
          -Converts an unsigned hex character string into its
          integer equivalent. !! Warning !! The function can and
          does modify the pointer to the input string parameter!
          The max string length the function assumes is four (4)
          for conversion to regular hexidecimal integer.


     interface:
          (p) str
          (p) validnum
          (p) intptr

     called by:
          get_inst()/debugopr.c
          get_mop()/debugopr.c
          get_dop()/debugopr.c
          get_rop()/debugopr.c
          get_bop()/debugopr.c

     calls:

     errors:
          -Initializes parameter 'validnum' with the results of
          of the conversion.

*/

str2hex(str,validnum,intptr)
char *str;
BOOLN *validnum;
long *intptr;

{    int index = 0;
     int maxchars;

     str = stripblk(str);
```

```
    if (strlen(str) == 0) {
            *validnum = FALSE;
            return;
    }

    *intptr = 0;

    maxchars = MXHEXSTR - 1;

    while ((((str[index] >= '0') && (str[index] <= '9')) ||
            ((str[index] >= 'a') && (str[index] <= 'f'))) &&
            (index < maxchars)) {

            if ((str[index] >= '0') && (str[index] <= '9'))
                    *intptr= 16 * (*intptr) + str[index++] - '0';
            else
                    *intptr= 16 * (*intptr) + (str[index++] - 'a') + 10;
    }

    str = &str[index];
    str = stripblk(str);
    *validnum = (strlen(str) == 0) ? TRUE : FALSE;
}
/*STR2DEC()
    function:
            -Converts an signed or unsigned character string into
            its long integer equivalent. !! Warning !! The function
            can and does modify the pointer to the input string para-
            meter! The max string length the function assumes is
            eleven (11) for conversion to regular signed integer.


    interface:
            (p) str
            (p) validnum
            (p) intptr

    called by:
            get_segm()/debugopr.c
            get_ofst()/debugopr.c
            getrange()/debugopr.c
            goexec()/debugopr.c
            traceop()/debugopr.c
            removbrk()/debugopr.c
            get_int()/debugopr.c
            get_nat()/debugopr.c
            get_file()/debugopr.c
            getvalue()/debugopr.c

    calls:
            strlen()/ *system*
```

```
        errors:
                -initializes parameter 'validnum' with the results of
                of the conversion.

*/

str2dec(str,validnum,intptr)
char *str;
BOOLN *validnum;
long *intptr;

{       int index = 0;
        int maxchars;
        int strsize;
        BOOLN negnum = FALSE;
        long maxnum = 2147483647;

        maxchars = MXDECSTR - 2;

        if (strlen(str) == 0) {
                *validnum = FALSE;
                return;
        }

        if (str[0] == '-') {
                negnum = TRUE;
                str = &str[1];
        }

        if (strlen(str) == 0) {
                *validnum = FALSE;
                return;
        }

        *intptr = 0;
        index = 0;

        while (((str[index] >= '0') && (str[index] <= '9'))
                 && (index < maxchars) && (*intptr <= maxnum)) {

                *intptr = 10 * (*intptr) + (str[index++] - '0');

        }

        str = &str[index];
        str = stripblk(str);
        *validnum = (strlen(str) == 0) ? TRUE : FALSE;

        if (*validnum) {
                *intptr = negnum ? 0 - *intptr : *intptr;
        }
}
```

LIST OF REFERENCES

Hunter, J. E., The Formal Specification of a Visual Display Device: Design and Implementation , Master's Thesis, Naval Postgraduate School, Monterey, Ca., Jun 1985.

Ozisik, M. G., Design and Implementation of a C Compiler for an Abstract Machine, Master's Thesis, Naval Postgraduate School, Monterey, Ca., Jun 1986.

Wray, B., Crawford, B., What Every Engineer Should Know About Microcomputer Systems Design and Debugging, Marcel Dekker, Inc., New York, N. Y., 1984, pp. 98 - 108.

Yurchak, J., The Formal Specification of an Abstract Machine, Design and Implementation , Master's Thesis, Naval Postgraduate School, Monterey, Ca., Dec 1984.

Zang, K. H., The Formal Specification of an Abstract Database: Design and Implementation , Master's Thesis, Naval Postgraduate School, Monterey, Ca., Dec 1985.

Naval Postgraduate School, Tech. Report NPS52 84-022, A Formal Method for Specifying Computer Resources in an Implementation Independent Manner, by Davis, D. L., Monterey, Ca., Nov 1984.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Chief of Naval Operations Director, Information Systems (OP-945) Navy Department Washington, D. C. 20350-2000	1
2.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
3.	Superintendent Attn: Library (Code 0142) Naval Postgraduate School Monterey, California 93943-5002	2
4.	Chairman (Code 52) Department of Computer Sciences Naval Postgraduate School Monterey, California 93943-5000	1
5.	Computer Technology Programs (Code 37) Naval Postgraduate School Monterey, California 93943-5000	1
6.	Daniel L. Davis (Code 52) Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	5
7.	Bruce J. MacLennan (Code 52) Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	5

8. 1stLt Stanley Victrum, USMC
c/o George Wilson 10
Rte 2, Box 116
Cross, South Carolina 29436

Thesis
V652 Victrum
c.1 Design and implementation of a debugger for an abstract machine.